

目录

序言	1	5.5 LINUX 的系统调用	160
本书的主要目标	1	5.6 系统时间和定时	162
现有书籍不足之处	1	5.7 LINUX 进程控制	164
阅读早期内核的其他好处	2	5.8 LINUX 系统中堆栈的使用方法	172
阅读完整源代码的重要性和必要性	2	5.9 LINUX 0.11 采用的文件系统	176
如何选择要阅读的内核代码版本	3	5.10 LINUX 内核源代码的目录结构	177
阅读本书需具备的基础知识	3	5.11 内核系统与应用程序的关系	184
使用早期版本是否过时?	4	5.12 LINUX/MAKEFILE 文件	184
EXT2 文件系统与 MINIX 文件系统	4	5.13 本章小结	189
第 1 章 概述	5	第 6 章 引导启动程序 (BOOT)	191
1.1 LINUX 的诞生和发展	5	6.1 总体功能	191
1.2 内容综述	12	6.2 BOOTSECT.S 程序	193
1.3 本章小结	16	6.3 SETUP.S 程序	203
第 2 章 微型计算机组成结构	17	6.4 HEAD.S 程序	221
2.1 微型计算机组成原理	17	6.5 本章小结	234
2.2 I/O 端口寻址和访问控制方式	19	第 7 章 初始化程序 (INIT)	235
2.3 主存储器、BIOS 和 CMOS 存储器	21	7.1 MAIN.C 程序	235
2.4 控制器和控制卡	23	7.2 环境初始化工作	247
2.5 本章小结	31	7.3 本章小结	249
第 3 章 内核编程语言和环境	32	第 8 章 内核代码 (KERNEL)	251
3.1 AS86 汇编器	32	8.1 总体功能	251
3.2 GNU AS 汇编	38	8.2 MAKEFILE 文件	254
3.3 C 语言程序	48	8.3 ASM.S 程序	256
3.4 C 与汇编程序的相互调用	55	8.4 TRAPS.C 程序	262
3.5 LINUX 0.11 目标文件格式	63	8.5 SYSTEM_CALL.S 程序	267
3.6 MAKE 程序和 MAKEFILE 文件	72	8.6 MKTIME.C 程序	279
第 4 章 80X86 保护模式及其编程	75	8.7 SCHED.C 程序	281
4.1 80X86 系统寄存器和系统指令	75	8.8 SIGNAL.C 程序	300
4.2 保护模式内存管理	81	8.9 EXIT.C 程序	311
4.3 分段机制	85	8.10 FORK.C 程序	318
4.4 分页机制	96	8.11 SYS.C 程序	326
4.5 保护	99	8.12 VSPRINTF.C 程序	333
4.6 中断和异常处理	110	8.13 PRINTK.C 程序	341
4.7 任务管理	120	8.14 PANIC.C 程序	342
4.8 保护模式编程初始化	128	8.15 本章小结	343
4.9 一个简单的多任务内核实例	131	第 9 章 块设备驱动程序 (BLOCK DRIVER)	345
第 5 章 LINUX 内核体系结构	141	9.1 总体功能	346
5.1 LINUX 内核模式	141	9.2 MAKEFILE 文件	349
5.2 LINUX 内核系统体系结构	142	9.3 BLK.H 文件	351
5.3 LINUX 内核对内存的管理和使用	144	9.4 HD.C 程序	355
5.4 中断机制	157	9.5 LL_RW_BLK.C 程序	378
		9.6 RAMDISK.C 程序	384
		9.7 FLOPPY.C 程序	390

第 10 章 字符设备驱动程序(CHAR DRIVER)	417	14.12 TIME.H 文件.....	722
10.1 总体功能	417	14.13 UNISTD.H 文件.....	724
10.2 MAKEFILE 文件.....	427	14.14 UTIME.H 文件	729
10.3 KEYBOARD.S 程序	429	14.15 INCLUDE/ASM/目录下的文件	731
10.4 CONSOLE.C 程序.....	448	14.16 IO.H 文件.....	731
10.5 SERIAL.C 程序	474	14.17 MEMORY.H 文件.....	732
10.6 RS_IO.S 程序	483	14.18 SEGMENT.H 文件.....	733
10.7 TTY_IO.C 程序.....	487	14.19 SYSTEM.H 文件.....	735
10.8 TTY_IOCTL.C 程序.....	499	14.20 INCLUDE/LINUX/目录下的文件	739
第 11 章 数学协处理器(MATH).....	507	14.21 CONFIG.H 文件	739
11.1 MAKEFILE 文件.....	507	14.22 FDREG.H 头文件	741
11.2 MATH-EMULATION.C 程序.....	509	14.23 FS.H 文件.....	744
第 12 章 文件系统(FS).....	511	14.24 HDREG.H 文件.....	749
12.1 总体功能	511	14.25 HEAD.H 文件	752
12.2 MAKEFILE 文件.....	527	14.26 KERNEL.H 文件.....	753
12.3 BUFFER.C 程序	530	14.27 MM.H 文件.....	754
12.4 BITMAP.C 程序.....	547	14.28 SCHED.H 文件.....	754
12.5 TRUNCTC.C 程序.....	553	14.29 SYS.H 文件	761
12.6 INODE.C 程序	555	14.30 TTY.H 文件.....	763
12.7 SUPER.C 程序	567	14.31 INCLUDE/SYS/目录中的文件.....	766
12.8 NAMEI.C 程序	577	14.32 STAT.H 文件	766
12.9 FILE_TABLE.C 程序.....	601	14.33 TIMES.H 文件.....	767
12.10 BLOCK_DEV.C 程序	601	14.34 TYPES.H 文件.....	768
12.11 FILE_DEV.C 程序.....	605	14.35 UTSNAME.H 文件.....	769
12.12 PIPE.C 程序.....	608	14.36 WAIT.H 文件.....	770
12.13 CHAR_DEV.C 程序	612	第 15 章 库文件(LIB).....	773
12.14 READ_WRITE.C 程序.....	615	15.1 MAKEFILE 文件	774
12.15 OPEN.C 程序	621	15.2 _EXIT.C 程序	776
12.16 EXEC.C 程序	627	15.3 CLOSE.C 程序	777
12.17 STAT.C 程序	647	15.4 CTYPE.C 程序	777
12.18 FCNTL.C 程序	649	15.5 DUP.C 程序	778
12.19 IOCTL.C 程序.....	652	15.6 ERRNO.C 程序.....	779
第 13 章 内存管理(MM).....	655	15.7 EXECVE.C 程序.....	779
13.1 总体功能	655	15.8 MALLOC.C 程序	780
13.2 MAKEFILE 文件.....	661	15.9 OPEN.C 程序	789
13.3 MEMORY.C 程序.....	662	15.10 SETSID.C 程序.....	790
13.4 PAGE.S 程序.....	679	15.11 STRING.C 程序.....	791
第 14 章 头文件(INCLUDE)	683	15.12 WAIT.C 程序.....	791
14.1 INCLUDE/目录下的文件	683	15.13 WRITE.C 程序	792
14.2 A.OUT.H 文件.....	684	第 16 章 建造工具(TOOLS).....	795
14.3 CONST.H 文件	695	16.1 BUILD.C 程序.....	795
14.4 CTYPE.H 文件	695	第 17 章 实验环境设置与使用方法	802
14.5 ERRNO.H 文件	697	17.1 BOCHS 仿真系统	802
14.6 FCNTL.H 文件	699	17.2 在 BOCHS 中运行 LINUX 0.11 系统.....	806
14.7 SIGNAL.H 文件	701	17.3 访问磁盘映像文件中的信息.....	813
14.8 STDARG.H 文件.....	703	17.4 编译运行简单内核示例程序.....	815
14.9 STDDEF.H 文件	704	17.5 利用 BOCHS 调试内核.....	817
14.10 STRING.H 文件	705	17.6 创建磁盘映像文件.....	824
14.11 TERMIOS.H 文件	715	17.7 制作根文件系统.....	827
		17.8 在 LINUX 0.11 系统上编译 0.11 内核.....	834
		17.9 在 REDHAT 9 系统下编译 LINUX 0.11 内核 ..	835

17.10 内核引导启动+根文件系统组成的集成盘	838	附录 2 ASCII 码表	863
17.11 从硬盘启动：利用 SHOELACE 引导软件....	843	附录 3 常用 C0、C1 控制字符表.....	864
17.12 利用 GDB 和 BOCHS 调试内核源代码	846	附录 4 常用转义序列和控制序列	865
参考文献.....	853	附录 5 第 1 套键盘扫描码集.....	868
附录.....	855	索引	869
附录 1 内核数据结构.....	855		

```

352 // 取父进程号 ppid。
353 int sys_getppid(void)
354 {
355     return current->father;
356 }
357 // 取用户号 uid。
358 int sys_getuid(void)
359 {
360     return current->uid;
361 }
362 // 取有效的用户号 euid。
363 int sys_geteuid(void)
364 {
365     return current->euid;
366 }
367 // 取组号 gid。
368 int sys_getgid(void)
369 {
370     return current->gid;
371 }
372 // 取有效的组号 egid。
373 int sys_getegid(void)
374 {
375     return current->egid;
376 }
377 // 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！！
378 int sys_nice(long increment)
379 {
380     if (current->priority-increment>0)
381         current->priority -= increment;
382     return 0;
383 }
384 // 内核调度程序的初始化子程序。
385 void sched_init(void)
386 {
387     int i;
388     struct desc_struct * p; // 描述符表结构指针。
389     // 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。
390     if (sizeof(struct sigaction) != 16) // sigaction 是存放有关信号状态的结构。
391         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]（即为 gdt[4]），也即 gdt 数组第 4 项的地址。参见
// include/asm/system.h, 第 65 行开始。
392     set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
393     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。
394     p = gdt+2+FIRST_TSS_ENTRY;
395     for(i=1;i<NR_TASKS;i++) {
396         task[i] = NULL;
397         p->a=p->b=0;
398         p++;
399         p->a=p->b=0;
400         p++;
401     }
402     /* 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 */
// NT 标志用于控制程序的递归调用(Nested Task)。当 NT 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。
403     __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 ldt。注意！！是将 GDT 中相应 LDT 描述符的选择符加载到 ldt。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。
404     ltr(0);
405     lldt(0);
// 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。
406     outb_p(0x36, 0x43); // binary, mode 3, LSB/MSB, ch 0 */
407     outb_p(LATCH & 0xff, 0x40); // LSB */ // 定时值低字节。
408     outb_p(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。
409     set_intr_gate(0x20, &timer_interrupt);
410     outb(inb_p(0x21)&~0x01, 0x21);
411     set_system_gate(0x80, &system_call);
412 }
413

```

8.7.3 其他信息

8.7.3.1 软盘驱动器控制器

有关对软盘控制器进行编程的详细说明请参见第 6 章程序 floppy.c 后面的解说，这里仅对其作一简单介绍。在对软盘控制器进行编程时需要访问 4 个端口。这些端口分别对应控制器上一个或多个寄存器。对于 1.2M 的软盘控制器有以下一些端口。

表 8-2 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器(数字控制寄存器)

0x3f4	只读	FDC 主状态寄存器
0x3f5	读/写	FDC 数据寄存器
0x3f7	只读	数字输入寄存器
0x3f7	只写	磁盘控制寄存器(传输率控制)

数字输出端口（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。

FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0--8 字节的参数。

执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。

结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0--7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

8.7.3.2 可编程定时/计数控制器

1. Intel 8253（8254）芯片功能

Intel 8253（或 8254）是一个可编程定时/计数器（PIT - Programmable Interval Timer）芯片，用于解决计算机中通常碰到的时间控制问题，即在软件的控制下产生精确的时间延迟。该芯片提供了 3 个独立的 16 位计数器通道。每个通道可工作在不同的工作方式下，并且这些工作方式均可以使用软件来设置。8254 是 8253 的更新产品，主要功能基本一样，只是 8254 芯片增加了回读命令。在下面描述中我们用 8253 来代称 8253 和 8254 两种芯片，仅在它们功能有区别处再特别加以指出。

8253 芯片的编程相对来说比较简单，并且能够产生所希望的各种不同时间长度的延时。一个 8253（8254）芯片的结构框图见图 8-8 所示。

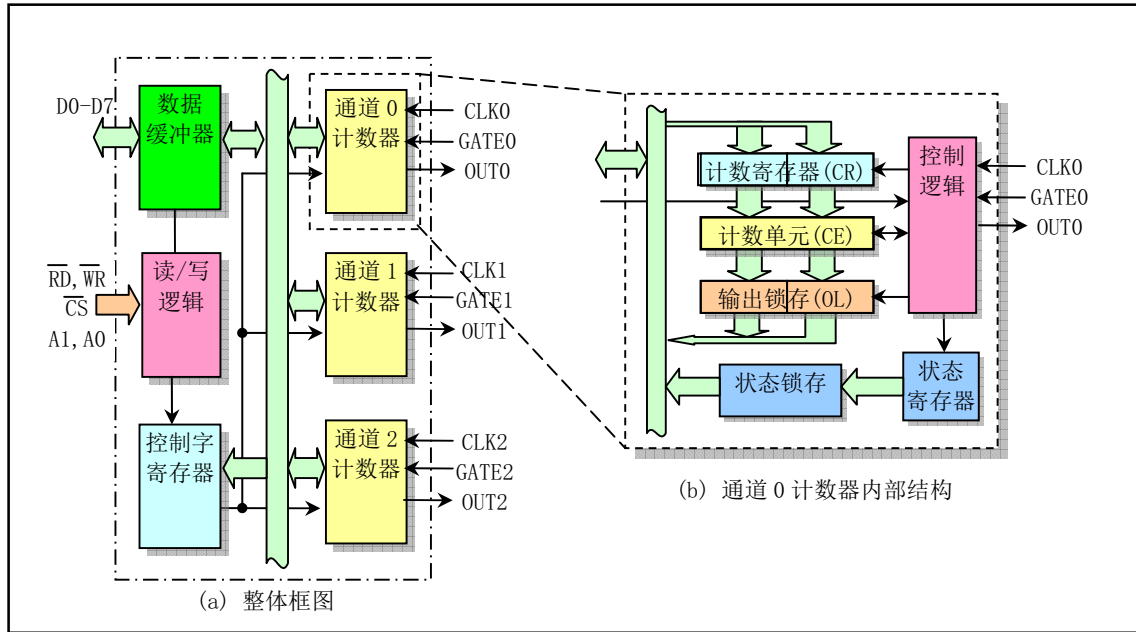


图 8-8 8253 (8254) 芯片的内部结构

其中 3 态、双向 8 位的数据总线缓冲器(Data Bus Buffer)用于与系统数据总线接口。读/写逻辑(Read/Write Logic)用于从系统总线上接收输入信号,并且生成输入到其他部分去的控制信号。地址线 A1、A0 用来选择需要读/写的 3 个计数器通道或控制字寄存器(Control Word Register)之一。通常它们被连接到系统的 A0,A1 地址线上。读写引脚 RD、WR 和片选引脚 CS 用于 CPU 控制 8253 芯片的读写操作。控制字寄存器用于 CPU 设置指定计数器的工作方式,是只写寄存器。但对于 8254 芯片则可以使用回读命令(Read-Back Command)来读取其中的状态信息。3 个独立的计数器通道作用完全相同,他们每个都可以工作在不同的方式下。控制字寄存器将确定每个计数器的工作方式。每个计数器的 CLK 引脚连接到时钟频率发生器(晶振)。8253 的时钟输入频率最高为 2.6MHz,而 8254 则可高达 10MHz。引脚 GATE 是计数器的门控制输入端,它用于控制计数器的起停以及计数器的输出状态。引脚 OUT 是计数器的输出信号端。

图 8-8(b)是其中一个计数器通道的内部逻辑框图。其中状态寄存器(Status Register)在锁定时将含有控制字寄存器的当前内容以及输出端的状态和 NULL 计数标志(Null Count Flag)。实际计数器是图中的 CE(计数单元)。它是一个 16 位可预置同步递减计数器。输出锁存 OL(Output Latch)是由 OLm 和 OLi 两个 8 位锁存器组成,分别表示锁存的高字节和低字节。通常这两个输出锁存器的内容跟随计数单元 CE 的内容变化而变化,但是如果芯片接收到一个计数器锁存命令时,那么它们的内容将被锁定。直到 CPU 读取它们的内容后,它们才会继续跟随 CE 的内容变化。注意,CE 的值是不可读的,每当你需要读取计数值时,读取的总是输出锁存 OL 中的内容。图 8-8(b)中另外两个是称为计数寄存器 CR(Count Register)的 8 位寄存器。当 CPU 把新的计数值写入计数器通道时,初始计数值就保存在这两个寄存器中,然后会被复制到计数单元 CE 中。在开始对计数器进行编程操作时,这两个寄存器将被清零。因此在初始计数值被保存到计数寄存器 CR 中之后,就会被送到计数单元 CE 中。当 GATE 开启时,计数单元会在时钟脉冲 CLK 的作用下执行递减计数操作。每次减 1,直到计数值递减为 0 时,就会向 OUT 引脚发出信号。

2. 8253 (8254) 芯片的编程

当系统刚上电时,8253 的状态是未知的。通过向 8253 写入一个控制字和一个初始计数值,我们就可以对想要使用的一个计数器进行编程。对于不使用的计数器我们可以不必对其编程。表 8-3 是控制寄

寄存器内容的格式。

表 8-3 8253 (8254) 芯片控制字格式

位	名称	说明
7	SC1	SC1、SC0 用于选择计数器通道 0-2，或者回读命令。(SC - Select Counter)
6	SC0	
5	RW1	RW1、RW0 用于计数器读写操作选择。(RW - Read Write)
4	RW0	
3	M2	M2-M0 用于选择指定通道的工作方式。(M - Method)
2	M1	
1	M0	
0	BCD	

在 CPU 执行写操作时，若 A1, A0 线为 11（此时在 PC 微机上对应端口 0x43），那么控制字会被写入控制字寄存器中。而控制字的内容会指定正在编程的计数器通道。初始计数值则会被写入指定的计数器中。当 A1, A0 为 00、01、10（分别对应 PC 机端口 0x40、0x41 和 0x42）时就会分别选择 3 个计数器之一。在写入操作时，必须首先写入控制字，然后再写入初始计数值。初始计数值必须根据控制字中设定的格式写入（二进制的或 BCD 码格式）。在计数器开始工作时，我们仍然能随时向指定计数器重新写入新的初始值。这并不会影响已设置的计数器的工作方式。

在读操作时，对于 8254 芯片可有 3 种方法来读取计数器的当前计数值：①简单读操作；②使用计数器锁存命令；③使用回读命令。第 1 种方法在读时必须使用 GATE 引脚或相应逻辑电路暂时停止计数器的时钟输入。否则计数操作可能正在进行，从而造成读取的结果有误。第 2 种方法是使用计数器锁存命令。该命令是在读操作前首先发送到控制字寄存器，并由 D5, D4 两个比特位（00）指明发送的是计数器锁存命令而非控制字命令。当计数器接收到该命令时，它会把计数单元 CE 中的计数值锁存到输出锁存寄存器 OL 中。此时 CPU 若不读取 OL 中的内容，那么 OL 中的数值将保持不变，即使你又发送了另外一条计数器锁存命令。只有在 CPU 执行了读取该计数器操作后，OL 的内容才又会自动地跟随计数单元 CE 进行变化。第 3 种方法是使用回读命令。但只有 8254 有此功能。这个命令允许程序检测当前计数值、计数器运行的方式，以及当前输出状态和 NULL 计数标志。与第 2 中方法类似，在锁定计数值后，只有在 CPU 执行了读取该计数器操作后，OL 的内容才又会自动地跟随计数单元 CE 进行变化。

3. 计数器工作方式

8253/8254 的 3 个计数器通道可以有各自独立的工作方式，有以下 6 种方式可供选择。

(1) 方式 0 - 计数结束中断方式 (Interrupt on terminal count)

该方式设定后，输出引脚 OUT 为低电平。并且始终保持为低电平直到计数递减为 0。此时 OUT 变为高电平并保持为高电平直到写入一个新的计数值或又重新设置控制字为方式 0。这种方式通常用于事件计数。这种方式的特点是允许使用 GATE 引脚控制计数暂停；计数结束时输出变高电平可作为中断信号；在计数期间可以重新装入初始计数值，并且在接收到计数高字节后重新执行计数工作。

(2) 方式 1 - 硬件可触发单次计数方式 (Hardware Retriggerable One-shot)

工作在这种方式下时，OUT 刚开始处于高电平。在 CPU 写入了控制字和初始计数值后，计数器准备就绪。此时可使用 GATE 引脚上升沿触发计数器开始工作，而 OUT 则变为低电平。直到计数结束 (0)，OUT 变为高电平。在计数期间或计数结束后，GATE 重新变高电平又会触发计数器装入初始计数值并重新开始计数操作。对于这种工作方式，GATE 信号不起作用。

(3) 方式 2 - 频率发生器方法 (Rate Generator)

该方式的功能类似于一个 N 分频器。通常用于产生实时时钟中断。初始状态下 OUT 为高电平。当计数值递减为 1 时，OUT 变为低电平后再变成高电平。间隔时间为一个 CLK 脉冲宽度。此时计数器会重新加载初始值并重复上述过程。因此对于初始计数值为 N 的情况，会在每 N 个时钟脉冲时输出一个低电平脉冲信号。在这种方式下 GATE 可控制计数的暂停和继续。当 GATE 变高时会让计数器重新加载初始值并开始重新计数。

(4) 方式 3 - 方波发生器方式 (Square Wave Mode)

该方式通常用于波特率发生器。该方式与方式 2 类似，但 OUT 输出的是方波。如果初始计数值是 N，那么方波的频率是输入时钟 CLK 的 N 分之一。该方式的特点是方波占空比约为 1 比 1（当 N 为奇数时略有差异），并且在计数器递减过程中若重新设置新的初始值，这个初始值要到前一个计数完成后才起作用。

(5) 方式 4 - 软件触发选通方式 (Software Triggered Strobe)

初始状态下 OUT 为高电平。当计数结束时 OUT 将输出一个时钟脉冲宽度的低电平，然后变高（低电平选通）。计数操作是由写入初始计数值而“触发”的。在该工作模式下，GATE 引脚可以控制计数暂停（1 允许计数），但不影响 OUT 的状态。如果在计数过程中写入了一个新的初始值，那么计数器会在一个时钟脉冲后使用新值来重新进行计数操作。

(6) 方式 5 - 硬件触发选通方式 (Hardware Triggered Strobe)

初始状态下 OUT 为高电平。计数操作将由 GATE 引脚上升沿触发。当计数结束，OUT 将输出一个时钟 CLK 脉冲宽度的低电平，然后变高。在写入控制字和初始值后，计数器并不会立刻加载初始计数值而开始工作。只有当 GATE 引脚变为高电平后的一个 CLK 时钟脉冲后才会被触发开始工作。

对于 PC/AT 及其兼容微机系统，采用的是 8254 芯片。3 个定时/计数器通道被分别用于日时钟计时中断信号、动态内存 DRAM 刷新定时电路和主机扬声器音调合成。3 个计数器的输入时钟频率都是 1.193180MHz。PC/AT 微机中 8254 芯片连接示意图见图 8-9 所示。其中 A1, A0 引线被连接到系统地址线 A1, A0 上。并且当系统地址线 A9--A2 信号是 0b0010000 时会选择 8254 芯片，因此 PC/AT 系统中 8254 芯片的 IO 端口地址范围是 0x40--0x43。其中 0x40--0x42 分别对应选择计数器通道 0--2，0x43 对应控制字寄存器写端口。

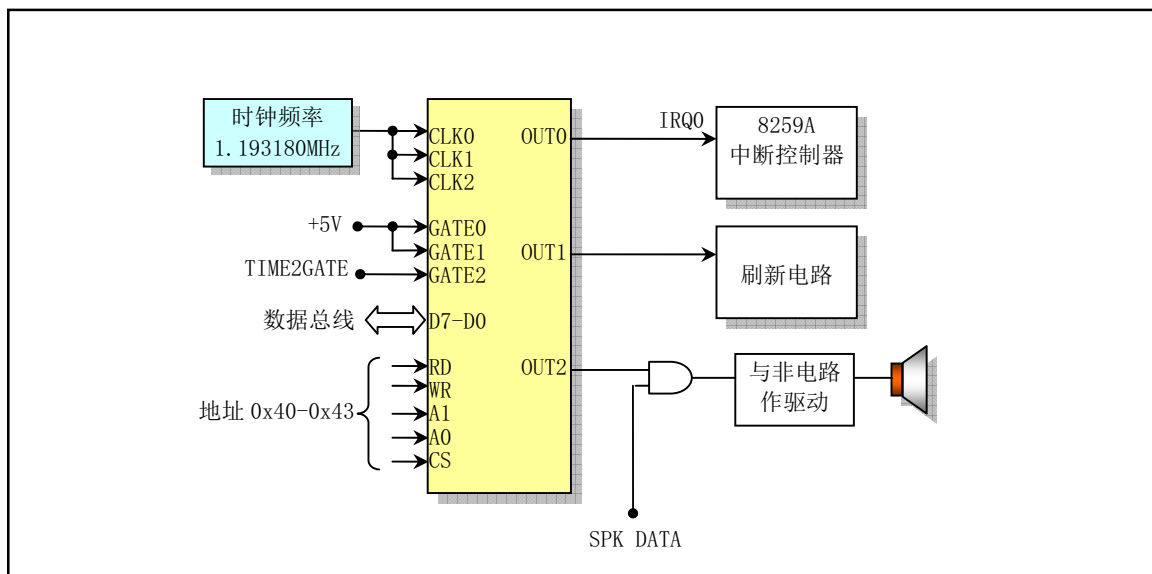


图 8-9 PC 微机中定时/计数芯片连接示意图

对于计数器通道 0，其 GATE 引脚固定连在高电平上。系统刚上电时它被 BIOS 程序设置成工作在方式 3 下（方波发生器方式），初始计数值被默认设置为 0，即 65536（0--65535）。因此每秒钟 OUT0 引脚会发出频率为 18.2HZ（ $1.193180\text{MHz}/65536$ ）的方波信号。OUT0 被连接至可编程中断控制器 8259 芯片的 0 级中断请求端。因此利用方波上升沿可触发中断请求，使得系统每间隔 54.9ms（ $1000\text{ms}/18.2$ ）就发出一个中断请求。

计数器通道 1 的 GATE 引脚也直接连接到高电平上，因此处于允许计数状态。其工作在方式 2（频率发生器方式）下，初始值通常被设置为 18。该计数器被用来向 PC/XT 系统的 DMA 控制器通道 2 或 PC/AT 系统的刷新电路发出 RAM 刷新信号。大约每 15 微秒输出一个信号，输出频率为 $1.19318/18 = 66.288\text{KHz}$ 。

计数器通道 2 的 GATE 引脚（TIME2GATE）被连接至 8255A 芯片端口 B 的 D0 引脚或等效逻辑电路中。图 8-9 中的 SPK DATA 被连接至 8255A 芯片端口 B（0x61）的 D1 引脚或等效逻辑电路中。该计数器通道用于让主机扬声器发出音调，但也可以与 8255A 芯片（或其等效电路）配合作为一个普通定时器使用。

Linux 0.11 操作系统只对 8254 的计数器通道 0 进行了重新设置，使得该计数器工作在方式 3 下、计数初始值采用二进制，并且初始计数值被设置为 LATCH（1193180/100）。即让计数器 0 每间隔 10 毫秒发出一个方波上升沿信号以产生中断请求信号（IRQ0）。因此向 8254 写入的控制字是 0x36（0b00110110），随后写入初始计数值的低字节和高字节。初始计数值低字节和高字节值分别为（LATCH & 0xff）和（LATCH >> 8）。这个间隔定时产生的中断请求就是 Linux 0.11 内核工作的脉搏，它用于定时切换当前执行的任务和统计每个任务使用的系统资源量（时间）。

8.8 signal.c 程序

8.8.1 功能描述

signal.c 程序涉及内核中所有有关信号处理的函数。在 UNIX 系统中，信号是一种“软件中断”处理机制。有许多较为复杂的程序会使用到信号。信号机制提供了一种处理异步事件的方法。例如，用户在终端键盘上键入 ctrl-C 组合键来终止一个程序的执行。该操作就会产生一个 SIGINT（SIGnal INterrupt）信号，并被发送到当前前台执行的进程中；当进程设置的一个报警时钟到期时，系统就会向进程发送一个 SIGALRM 信号；当发生硬件异常时，系统也会向正在执行的进程发送相应的信号。另外，一个进程也可以向另一个进程发送信号。例如使用 kill() 函数向同组的子进程发送终止执行信号。

信号处理机制在很早的 UNIX 系统中就已经有了，但那些早期 UNIX 内核中信号处理的方法并不是那么可靠。信号可能会被丢失，而且在处理紧要区域代码时进程有时很难关闭一个指定的信号，后来 POSIX 提供了一种可靠处理信号的方法。为了保持兼容性，本程序中还是提供了两种处理信号的方法。

在内核代码中通常使用一个无符号长整数（32 位）中的比特位来表示各种不同信号。因此最多可表示 32 个不同的信号。在本版 Linux 内核中，定义了 22 种不同的信号。其中 20 种信号是 POSIX.1 标准中规定的所有信号，另外 2 种是 Linux 的专用信号：SIGUNUSED（未定义）和 SIGSTKFLT（堆栈错），前者可表示系统目前还不支持的所有其他信号种类。这 22 种信号的具体名称和定义可参考程序后的信号列表，也可参阅 include/signal.h 头文件。

对于进程来说，当收到一个信号时，可以由三种不同的处理或操作方式。

1. 忽略该信号。大多数信号都可以被进程忽略。但有两个信号忽略不掉：SIGKILL 和 SIGSTOP。其原因是为了向超级用户提供一个确定的方法来终止或停止指定的任何进程。另外，若忽略掉某些硬件异常而产生的信号（例如被 0 除），则进程的行为或状态就可能变得不可知了。
2. 捕获该信号。为了进行该操作，我们必须首先告诉内核在指定的信号发生时调用我们自定义的信号处理函数。在该处理函数中，我们可以做任何操作，当然也可以什么不做，起到忽略该信号的同样

作用。自定义信号处理函数来捕获信号的一个例子是：如果我们在程序执行过程中创建了一些临时文件，那么我们就可以定义一个函数来捕获 SIGTERM（终止执行）信号，并在该函数中做一些清理临时文件的工作。SIGTERM 信号是 kill 命令发送的默认信号。

3. 执行默认操作。内核为每种信号都提供一种默认操作。通常这些默认操作就是终止进程的执行。参见程序后信号列表中的说明。

本程序给出了设置和获取进程信号阻塞码（屏蔽码）系统调用函数 `sys_ssetmask()` 和 `sys_sgetmask()`、信号处理系统调用 `sys_signal()`（即传统信号处理函数 `signal()`）、修改进程在收到特定信号时所采取的行动的系统调用 `sys_sigaction()`（既可靠信号处理函数 `sigaction()`）以及在系统调用中断处理程序中处理信号的函数 `do_signal()`。有关信号操作的发送信号函数 `send_sig()` 和通知父进程函数 `tell_father()` 则被包含在另一个程序（`exit.c`）中。程序中的名称前缀 `sig` 均是信号 `signal` 的简称。

`signal()` 和 `sigaction()` 的功能比较类似，都是更改信号原处理句柄（`handler`，或称为处理程序）。但 `signal()` 就是内核操作上述传统信号处理的方式，在某些特殊时刻可能会造成信号丢失。当用户想对特定信号使用自己的信号处理程序（信号句柄）时，需要使用 `signal()` 或 `sigaction()` 系统调用首先在进程自己的任务数据结构中设置 `sigaction[]` 结构数组项，把自身信号处理程序的指针和一些属性“记录”在该结构项中。当内核在退出一个系统调用和某些中断过程时会检测当前进程是否收到信号。若收到了用户指定的特定信号，内核就会根据进程任务数据结构中 `sigaction[]` 中对应信号的结构项执行用户自己定义的信号处理服务程序。

在 `include/signal.h` 头文件第 55 行上，`signal()` 函数原型声明如下：

```
void (*signal(int signr, void (*handler)(int)))(int);
```

这个 `signal()` 函数有两个参数。一个指定需要捕获的信号 `signr`；另外一个新的信号处理函数指针（新的信号处理句柄）`void (*handler)(int)`。新的信号处理句柄是一个无返回值且具有一个整型参数的函数指针，该整型参数用于当指定信号发生时内核将其传递给处理句柄。

`signal()` 函数的原型声明看上去比较复杂，但是若我们定义一个如下类型：

```
typedef void sigfunc(int);
```

那么我们可以把 `signal()` 函数的原型改写成下面的简单样子：

```
sigfunc *signal(int signr, sigfunc *handler);
```

`signal()` 函数会给信号值是 `signr` 的信号安装一个新的信号处理函数句柄 `handler`，该信号句柄可以是用户指定的一个信号处理函数，也可以是内核提供的特定的函数指针 `SIG_IGN` 或 `SIG_DFL`。

当指定的信号到来时，如果相关的信号处理句柄被设置成 `SIG_IGN`，那么该信号就会被忽略掉。如果信号句柄是 `SIG_DFL`，那么就会执行该信号的默认操作。否则，如果信号句柄被设置成用户的一个信号处理函数，那么内核首先会把该信号句柄被复位成其默认句柄，或者会执行与实现相关的信号阻塞操作，然后会调用执行指定的信号处理函数。

`signal()` 函数会返回原信号处理句柄，这个返回的句柄也是一个无返回值且具有一个整型参数的函数指针。并且在新句柄被调用执行过一次后，信号处理句柄又会被恢复成默认处理句柄值 `SIG_DFL`。

在 `include/signal.h` 文件中（第 45 行起），默认句柄 `SIG_DFL` 和忽略处理句柄 `SIG_IGN` 的定义是：

```
#define SIG_DFL      ((void (*)(int))0)
#define SIG_IGN      ((void (*)(int))1)
```

都分别表示无返回值的函数指针，与 `signal()` 函数中第二个参数的要求相同。指针值分别是 0 和 1。这两个指针值逻辑上讲是实际程序中不可能出现的函数地址值。因此在 `signal()` 函数中就可以根据这两个特殊的指针值来判断是否使用默认信号处理句柄或忽略对信号的处理（当然 `SIGKILL` 和 `SIGSTOP` 是不能被忽略的）。参见下面程序列表中第 94—98 行的处理过程。

当一个程序被执行时，系统会设置其处理所有信号的方式为 `SIG_DFL` 或 `SIG_IGN`。另外，当程序 `fork()` 一个子进程时，子进程会继承父进程的信号处理方式（信号屏蔽码）。因此父进程对信号的设置和处理方式在子进程中同样有效。

为了能连续地捕获一个指定的信号，`signal()` 函数的通常使用方式例子如下。

```
void sig_handler(int signr)           // 信号句柄。
{
    signal(SIGINT, sig_handler);      // 为处理下一次信号发生而重新设置自己的处理句柄。
    ...
}

main ()
{
    signal(SIGINT, sig_handler);      // 主程序中设置自己的信号处理句柄。
    ...
}
```

`signal()` 函数不可靠的原因在于当信号已经发生而进入自己设置的信号处理函数中，但在重新再一次设置自己的处理句柄之前，在这段时间内有可能又有一个信号发生。但是此时系统已经把处理句柄设置成默认值。因此就有可能造成信号丢失。

`sigaction()` 函数采用了 `sigaction` 数据结构来保存指定信号的信息，它是一种可靠的内核处理信号的机制，它可以让我们方便地查看或修改指定信号的处理句柄。该函数是 `signal()` 函数的一个超集。该函数在 `include/signal.h` 头文件（第 66 行）中的声明为：

```
int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
```

其中参数 `sig` 是我们需要查看或修改其信号处理句柄的信号，后两个参数是 `sigaction` 结构的指针。当参数 `act` 指针不是 `NULL` 时，就可以根据 `act` 结构中的信息修改指定信号的行为。当 `oldact` 不为空时，内核就会在该结构中返回信号原来的设置信息。`sigaction` 结构见如下所示：

```
48 struct sigaction {
49     void (*sa_handler)(int);           // 信号处理句柄。
50     sigset_t sa_mask;                  // 信号的屏蔽码，可以阻塞指定的信号集。
51     int sa_flags;                       // 信号选项标志。
52     void (*sa_restorer)(void);         // 信号恢复函数指针（系统内部使用）。
53 };
```

当修改一个信号的处理方法时，如果处理句柄 `sa_handler` 不是默认处理句柄 `SIG_DFL` 或忽略处理句柄 `SIG_IGN`，那么在 `sa_handler` 处理句柄可被调用前，`sa_mask` 字段就指定了需要加入到进程信号屏蔽位图中的一个信号集。如果信号处理句柄返回，系统就会恢复进程原来的信号屏蔽位图。这样在一个信号句柄被调用时，我们就可以阻塞指定的一些信号。当信号句柄被调用时，新的信号屏蔽位图会自动地把当前发送的信号包括进去，阻塞该信号的继续发送。从而在我们处理一指定信号期间能确保阻塞同一

个信号而不让其丢失，直到此次处理完毕。另外，在一个信号被阻塞期间而又多次发生时通常只保存其一个样例，也即在阻塞解除时对于阻塞的多个同一信号只会再调用一次信号处理句柄。在我们修改了一个信号的处理句柄之后，除非再次更改，否则就一直使用该处理句柄。这与传统的 `signal()` 函数不一样。`signal()` 函数会在一处理句柄结束后将其恢复成信号的默认处理句柄。

`sigaction` 结构中的 `sa_flags` 用于指定其他一些处理信号的选项，这些选项的定义请参见 `include/signal.h` 文件中（第 36-39 行）的说明。

`sigaction` 结构中的最后一个字段和 `sys_signal()` 函数的参数 `restorer` 是一函数指针。它在编译连接程序时由 Libc 函数库提供，用于在信号处理程序结束后清理用户态堆栈，并恢复系统调用存放在 `eax` 中的返回值，见下面详细说明。

`do_signal()` 函数是内核系统调用(`int 0x80`)中断处理程序中对信号的预处理程序。在进程每次调用系统调用或者发生时钟等中断时，若进程已收到信号，则该函数就会把信号的处理句柄（即对应的信号处理函数）插入到用户程序堆栈中。这样，在当前系统调用结束返回后就会立刻执行信号句柄程序，然后再继续执行用户的程序，见图 8-10 所示。

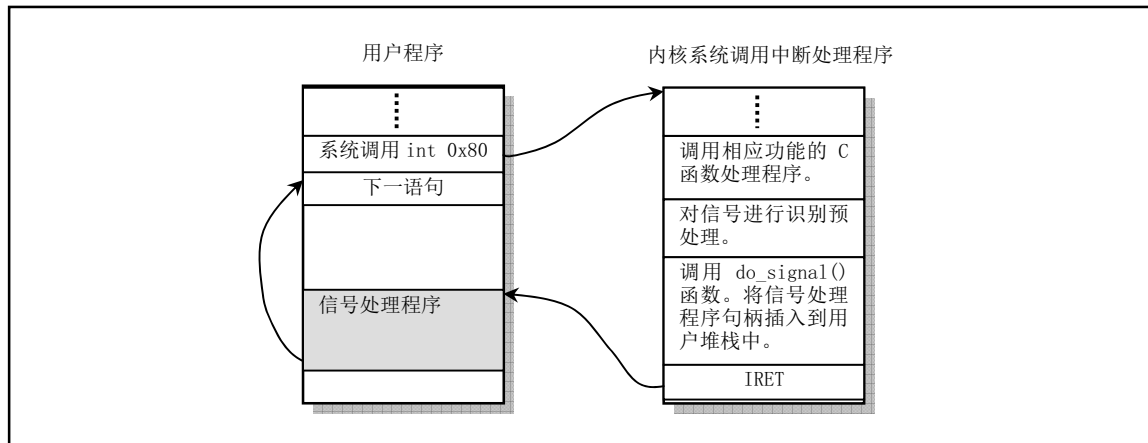


图 8-10 信号处理程序的调用方式。

在把信号处理程序的参数插入到用户堆栈中之前，`do_signal()` 函数首先会把用户程序堆栈指针向下扩展 `long` 个长字（参见下面程序中 106 行），然后将相关的参数添入其中，参见图 8-11 所示。由于 `do_signal()` 函数从 104 行开始的代码比较难以理解，下面我们将对其进行详细描述。

在用户程序调用系统调用刚进入内核时，该进程的内核态堆栈上会由 CPU 自动压入如图 8-11 中所示的内容，也即：用户程序的 `SS` 和 `ESP` 以及用户程序中下一条指令的执行点位置 `CS` 和 `EIP`。在处理完此次指定的系统调用功能并准备调用 `do_signal()` 时（也即 `system_call.s` 程序 118 行之后），内核态堆栈中的内容见图 8-12 中左边所示。因此 `do_signal()` 的参数即是这些在内核态堆栈上的内容。

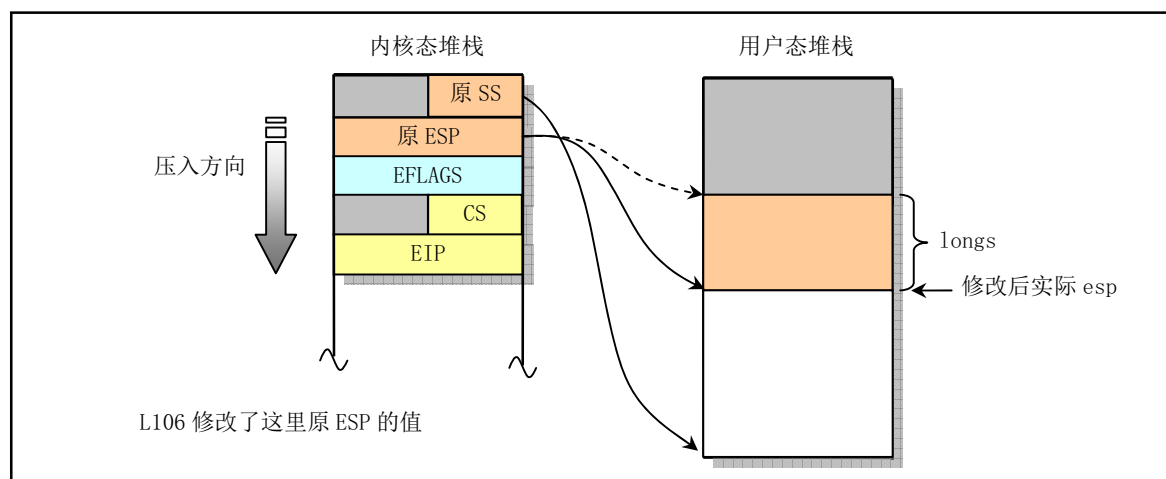


图 8-11 do_signal() 函数对用户堆栈的修改

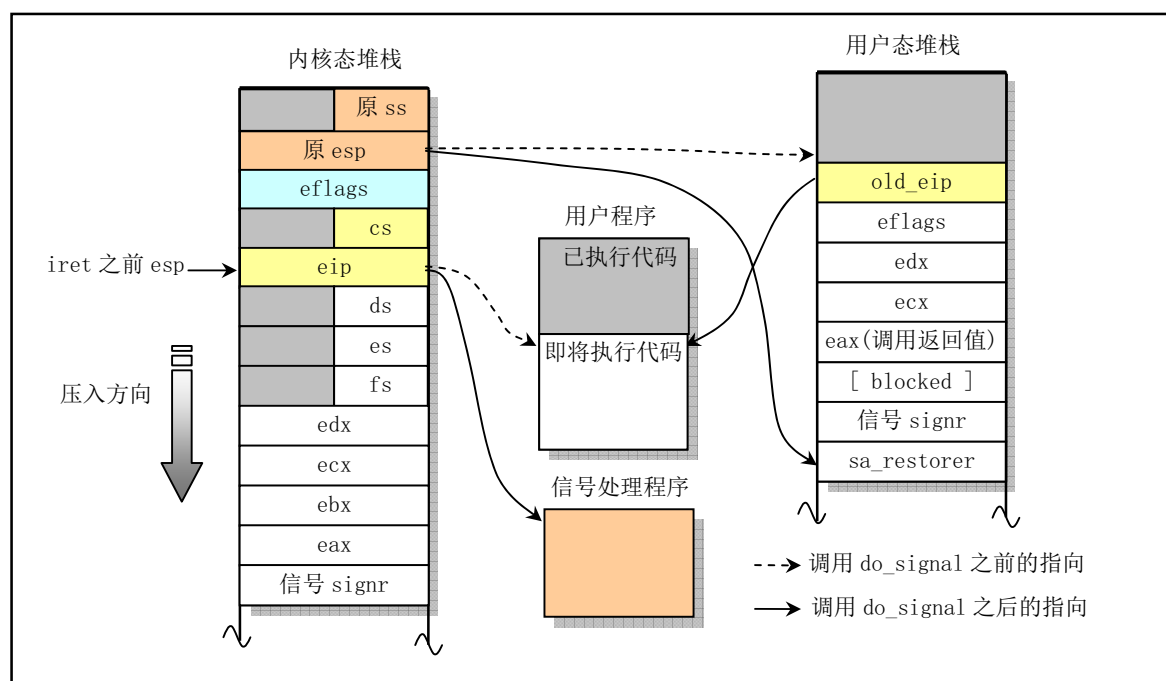


图 8-12 do_signal() 函数修改用户堆栈的具体过程

在 do_signal() 处理完两个默认信号句柄 (SIG_IGN 和 SIG_DFL) 之后, 若用户自定义了信号处理程序 (信号句柄 sa_handler), 则从 104 行起 do_signal() 开始准备把用户自定义的句柄插入用户态堆栈中。它首先把内核态堆栈中原用户程序的返回执行点指针 eip 保存为 old_eip, 然后将该 eip 替换成指向自定义句柄 sa_handler, 也即让图中内核态堆栈中的 eip 指向 sa_handler。接下来通过把内核态中保存的“原 esp”减去 longs 值, 把用户态堆栈向下扩展了 7 或 8 个长字空间。最后把内核堆栈上的一些寄存器内容复制到了这个空间中, 见图中右边所示。

总共往用户态堆栈上放置了 7 到 8 个值, 我们现在来说明这些值的含义以及放置这些值的原因。

old_eip 即是原用户程序的返回地址, 它是在内核堆栈上 eip 被替换成信号句柄地址之前保留下来的。eflags、edx 和 ecx 是原用户程序在调用系统调用之前的值, 基本上也是调用系统调用的参数, 在系统调用返回后仍然需要恢复这些用户程序的寄存器值。eax 中保存有系统调用的返回值。如果所处理的信号还允许收到本身, 则堆栈上还存放有该进程的阻塞码 blocked。下一个是信号 signr 值。

最后一个信号活动恢复函数的指针 `sa_restorer`。这个恢复函数不是由用户设定的，因为在用户定义 `signal()` 函数时只提供了一个信号值 `signr` 和一个信号处理句柄 `handler`。

下面是为 `SIGINT` 信号设置自定义信号处理句柄的一个简单例子，默认情况下，按下 `Ctrl-C` 组合键会产生 `SIGINT` 信号。

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

void handler(int sig)                // 信号处理句柄。
{
    printf("The signal is %d\n", sig);
    (void) signal(SIGINT, SIG_DFL); // 恢复 SIGINT 信号的默认处理句柄。（实际上内核会
}                                     // 自动恢复默认值，但对于其他系统未必如此）

int main()
{
    (void) signal(SIGINT, handler); // 设置 SIGINT 的用户自定义信号处理句柄。
    while (1) {
        printf("Signal test.\n");
        sleep(1);                    // 等待 1 秒钟。
    }
}
```

其中，信号处理函数 `handler()` 会在信号 `SIGINT` 出现时被调用执行。该函数首先输出一条信息，然后会把 `SIGINT` 信号的处理过程设置成默认信号处理句柄。因此在第二次按下 `Ctrl-C` 组合键时，`SIG_DFL` 会让该程序结束运行。

那么 `sa_restorer` 这个函数是从哪里来的呢？其实它是由函数库提供的。在 Linux 的 Libc 2.2.2 函数库文件（`misc/`子目录）中有它的函数，定义如下：

```
.globl __sig_restore
.globl __mask_sig_restore
# 若没有 blocked 则使用这个 restorer 函数
__sig_restore:
    addl $4,%esp        # 丢弃信号值 signr
    popl %eax            # 恢复系统调用返回值。
    popl %ecx            # 恢复原用户程序寄存器值。
    popl %edx
    popfl               # 恢复用户程序时的标志寄存器。
    ret
# 若有 blocked 则使用下面这个 restorer 函数，blocked 供 ssetmask 使用。
__mask_sig_restore:
    addl $4,%esp        # 丢弃信号值 signr
    call __ssetmask     # 设置信号屏蔽码 old blocking
    addl $4,%esp        # 丢弃 blocked 值。
    popl %eax
    popl %ecx
    popl %edx
    popfl
    ret
```

该函数的主要作用是为了在信号处理程序结束后，恢复用户程序执行系统调用后的返回值和一些寄存器内容，并清除作为信号处理程序参数的信号值 `signr`。在编译连接用户自定义的信号处理函数时，编译程序会调用 `Libc` 库中信号系统调用函数把 `sa_restorer()` 函数插入到用户程序中。库文件中信号系统调用的函数实现见如下所示。

```

01 #define __LIBRARY__
02 #include <unistd.h>
03
04 extern void __sig_restore();
05 extern void __mask_sig_restore();
06
    // 库函数中用户调用的 signal() 包裹函数。
07 void (*signal(int sig, __sig_handler_t func))(int)
08 {
09     void (*res)();
10     register int __foebx __asm__ ("bx") = sig;
11     __asm__ ("int $0x80": "=a" (res):
12         "0" (__NR_signal), "r" (__foebx), "c" (func), "d" ((long) __sig_restore));
13     return res;
14 }
15
    // 用户调用的 sigaction() 函数。
16 int sigaction(int sig, struct sigaction * sa, struct sigaction * old)
17 {
18     register int __foebx __asm__ ("bx") = sig;
19     if (sa->sa_flags & SA_NOMASK)
20         sa->sa_restorer = __sig_restore;
21     else
22         sa->sa_restorer = __mask_sig_restore;
23     __asm__ ("int $0x80": "=a" (sig)
24         : "0" (__NR_sigaction), "r" (__foebx), "c" (sa), "d" (old));
25     if (sig >= 0)
26         return 0;
27     errno = -sig;
28     return -1;
29 }

```

`sa_restorer()` 函数负责清理在信号处理程序执行完后恢复用户程序的寄存器值和系统调用返回值，就好像没有运行过信号处理程序，而是直接从系统调用中返回的。

最后说明一下执行的流程。在 `do_signal()` 执行完后，`system_call.s` 将会把进程内核态堆栈上 `eip` 以下的值弹出堆栈。在执行了 `iret` 指令之后，CPU 会把内核态堆栈上的 `cs:eip`、`eflags` 以及 `ss:esp` 弹出，恢复到用户态去执行程序。由于 `eip` 已经被替换为指向信号句柄，因此，此刻即会立即执行用户自定义的信号处理程序。在该信号处理程序执行完后，通过 `ret` 指令，CPU 会把控制权移交给 `sa_restorer` 所指向的恢复程序去执行。而 `sa_restorer` 程序会做一些用户态堆栈的清理工作，也即会跳过堆栈上的信号值 `signr`，并把系统调用后的返回值 `eax` 和寄存器 `ecx`、`edx` 以及标志寄存器 `eflags` 弹出，完全恢复了系统调用后各寄存器和 CPU 的状态。最后通过 `sa_restorer` 的 `ret` 指令弹出原用户程序的 `eip`（也即堆栈上的 `old_eip`），返回去执行用户程序。

8.8.2 代码注释

程序 8-7 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。
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，信号结构以及信号操作函数原型。
12
13 // 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
14 // 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。
15 // 等同于现在 gcc 的函数属性说明：void do_exit(int error_code) __attribute__((noreturn));
16 volatile void do_exit(int error_code);
17
18 // 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。
19 int sys_sgetmask()
20 {
21     return current->blocked;
22 }
23
24 // 设置新的信号屏蔽位图。SIGKILL 不能被屏蔽。返回值是原信号屏蔽位图。
25 int sys_ssetmask(int newmask)
26 {
27     int old=current->blocked;
28
29     current->blocked = newmask & ~(1<<(SIGKILL-1));
30     return old;
31 }
32
33 // 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。
34 static inline void save_old(char * from, char * to)
35 {
36     int i;
37
38     // 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）
39     // 空间中。宏函数 put_fs_byte() 在 include/asm/segment.h 中实现。
40     verify_area(to, sizeof(struct sigaction));
41     for (i=0 ; i< sizeof(struct sigaction) ; i++) {
42         put_fs_byte(*from, to);
43         from++;
44         to++;
45     }
46 }
47
48 // 把 sigaction 数据从 fs 数据段 from 位置复制到 to 处。即从用户数据空间复制到内核数据段中。

```

```

40 static inline void get\_new(char * from, char * to)
41 {
42     int i;
43
44     for (i=0 ; i< sizeof(struct sigaction) ; i++)
45         *(to++) = get\_fs\_byte(from++);
46 }
47
// 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，就好像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。
48 int sys\_signal(int signum, long handler, long restorer)
49 {
50     struct sigaction tmp;
51
// 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。
52     if (signum<1 || signum>32 || signum==SIGKILL)
53         return -1;
// 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。
54     tmp.sa_handler = (void (*)(int)) handler;
55     tmp.sa_mask = 0;
56     tmp.sa_flags = SA\_ONESHOT | SA\_NOMASK;
57     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。
58     handler = (long) current->sigaction[signum-1].sa_handler;
59     current->sigaction[signum-1] = tmp;
60     return handler;
61 }
62
// 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-1。
63 int sys\_sigaction(int signum, const struct sigaction * action,
64                   struct sigaction * oldaction)
65 {
66     struct sigaction tmp;
67
// 信号值要在 (1-32) 范围内，并且信号 SIGKILL 的处理句柄不能被改变。
68     if (signum<1 || signum>32 || signum==SIGKILL)
69         return -1;
// 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。
70     tmp = current->sigaction[signum-1];
71     get\_new((char *) action,
72            (char *) (signum-1+current->sigaction));
73     if (oldaction)
74         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。
75     if (current->sigaction[signum-1].sa_flags & SA\_NOMASK)
76         current->sigaction[signum-1].sa_mask = 0;
77     else
78         current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
79     return 0;
80 }
81
// 系统调用的中断处理程序中真正的信号预处理程序 (在 kernel/system_call.s, 119 行)。
// 该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。这个函数处理比较粗略, 尚不能处
// 理进程暂停 SIGSTOP 等信号。
// 函数的参数是进入系统调用处理程序 system_call.s 开始, 直到调用本函数 (system_call.s
// 第 119 行) 前逐步压入堆栈的值。这些值包括 (在 system_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 83--88 行在刚进入 system_call 时压入栈的寄存器 ds、es、fs 和 edx、ecx、ebx;
// ③ 第 95 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 118 行压入栈中的当前处理的信号值 (signr)。
82 void do\_signal(long signr, long eax, long ebx, long ecx, long edx,
83     long fs, long es, long ds,
84     long eip, long cs, long eflags,
85     unsigned long * esp, long ss)
86 {
87     unsigned long sa_handler;
88     long old_eip=eip;
89     struct sigaction * sa = current->sigaction + signr - 1;
90     int longs;                                // 即 current->sigaction[signr-1]。
91     unsigned long * tmp_esp;
92
93     // 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; 如果句柄为
94     // SIG_DFL (0, 默认处理), 则如果信号是 SIGCHLD 也直接返回, 否则终止进程的执行。
95     // 句柄 SIG_IGN 被定义为 1, SIG_DFL 被定义为 0。参见 include/signal.h, 第 45、46 行。
96     // 第 100 行 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
97     // 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
98     // 进程的退出状态码或子进程终止的原因 (信号)。
99     sa_handler = (unsigned long) sa->sa_handler;
100    if (sa_handler==1)
101        return;
102    if (!sa_handler) {
103        if (signr==SIGCHLD)
104            return;
105        else
106            do\_exit(1<<(signr-1));                // 不再返回到这里。
107    }
108    // OK, 以下准备对信号句柄的调用设置。 如果该信号句柄只需使用一次, 则将该句柄置空。
109    // 注意, 该信号句柄已经保存在 sa_handler 指针中。
110    // 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
111    // 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
112    // sa_restorer、signr、进程屏蔽码 (如果 SA_NOMASK 没置位)、eax、ecx、edx 作为参数以及
113    // 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
114    // 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。
115    if (sa->sa_flags & SA\_ONESHOT)
116        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 104 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，则也需要将进程的阻塞码压入堆栈。
104     *(&eip) = sa_handler;
105     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）。
106     *(&esp) -= longs;
107     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。
108     tmp_esp=esp;
109     put_fs_long((long) sa->sa_restorer, tmp_esp++);
110     put_fs_long(signr, tmp_esp++);
111     if (!(sa->sa_flags & SA_NOMASK))
112         put_fs_long(current->blocked, tmp_esp++);
113     put_fs_long(eax, tmp_esp++);
114     put_fs_long(ecx, tmp_esp++);
115     put_fs_long(edx, tmp_esp++);
116     put_fs_long(eflags, tmp_esp++);
117     put_fs_long(old_eip, tmp_esp++);
118     current->blocked |= sa->sa_mask; // 进程阻塞码(屏蔽码)添上 sa_mask 中的码位。
119 }
120

```

8.8.3 其他信息

8.8.3.1 进程信号说明

进程中的信号是用于进程之间通信的一种简单消息，通常是下表中的一个标号数值，并且不携带任何其他的信息。例如当一个子进程终止或结束时，就会产生一个标号为 18 的 SIGCHLD 信号发送给父进程，以通知父进程有关子进程的当前状态。

关于一个进程如何处理收到的信号，一般有两种做法：一是程序的进程不去处理，此时该信号会由系统相应的默认信号处理程序进行处理；第二种做法是进程使用自己的信号处理程序来处理信号。Linux 0.11 内核所支持的信号见表 8-4 所示。

表 8-4 进程信号

标号	名称	说明	默认操作
1	SIGHUP	(Hangup) 当你不再控制终端时内核会产生该信号，或者当你关闭 Xterm 或断开 modem。由于后台程序没有控制的终端，因而它们常用 SIGHUP 来发出需要重新读取其配置文件的信号。	(Abort) 挂断控制终端或进程。
2	SIGINT	(Interrupt) 来自键盘的中断。通常终端驱动程序会将其与 ^C 绑定。	(Abort) 终止程序。
3	SIGQUIT	(Quit) 来自键盘的退出中断。通常终端驱动程序会将其与 ^\ 绑定。	(Dump) 程序被终止并

			产生 dump core 文件。
4	SIGILL	(Illegal Instruction) 程序出错或者执行了一条非法操作指令。	(Dump) 程序被终止并产生 dump core 文件。
5	SIGTRAP	(Breakpoint/Trace Trap) 调试用, 跟踪断点。	
6	SIGABRT	(Abort) 放弃执行, 异常结束。	(Dump) 程序被终止并产生 dump core 文件。
7	SIGIOT	(IO Trap) 同 SIGABRT	(Dump) 程序被终止并产生 dump core 文件。
8	SIGUNUSED	(Unused) 没有使用。	
9	SIGFPE	(Floating Point Exception) 浮点异常。	(Dump) 程序被终止并产生 dump core 文件。
10	SIGKILL	(Kill) 程序被终止。该信号不能被捕获或者被忽略。想立刻终止一个进程, 就发送信号 9。注意程序将没有任何机会做清理工作。	(Abort) 程序被终止。
11	SIGUSR1	(User defined Signal 1) 用户定义的信号。	(Abort) 进程被终止。
12	SIGSEGV	(Segmentation Violation) 当程序引用无效的内存时会产生此信号。比如: 寻址没有映射的内存; 寻址未许可的内存。	(Dump) 程序被终止并产生 dump core 文件。
13	SIGUSR2	(User defined Signal 2) 保留给用户程序用于 IPC 或其他目的。	(Abort) 进程被终止。
14	SIGPIPE	(Pipe) 当程序向一个套接字或管道写时由于没有读者而产生该信号。	(Abort) 进程被终止。
15	SIGALRM	(Alarm) 该信号会在用户调用 alarm 系统调用所设置的延迟时间到后产生。该信号常用于判别系统调用超时。	(Abort) 进程被终止。
16	SIGTERM	(Terminate) 用于和善地要求一个程序终止。它是 kill 的默认信号。与 SIGKILL 不同, 该信号能被捕获, 这样就能在退出运行前做清理工作。	(Abort) 进程被终止。
17	SIGSTKFLT	(Stack fault on coprocessor) 协处理器堆栈错误。	(Abort) 进程被终止。
18	SIGCHLD	(Child) 子进程发出。子进程已停止或终止。可改变其含义挪作它用。	(Ignore) 子进程停止或结束。
19	SIGCONT	(Continue) 该信号致使被 SIGSTOP 停止的进程恢复运行。可以被捕获。	(Continue) 恢复进程的 执行。
20	SIGSTOP	(Stop) 停止进程的运行。该信号不可被捕获或忽略。	(Stop) 停止进程运行。
21	SIGTSTP	(Terminal Stop) 向终端发送停止键序列。该信号可以被捕获或忽略。	(Stop) 停止进程运行。
22	SIGTTIN	(TTY Input on Background) 后台进程试图从一个不再被控制的终端上读取数据, 此时该进程将被停止, 直到收到 SIGCONT 信号。该信号可以被捕获或忽略。	(Stop) 停止进程运行。
23	SIGTTOU	(TTY Output on Background) 后台进程试图向一个不再被控制的终端上输出数据, 此时该进程将被停止, 直到收到 SIGCONT 信号。该信号可被捕获或忽略。	(Stop) 停止进程运行。

8.9 exit.c 程序

8.9.1 功能描述

该程序主要描述了进程（任务）终止和退出的有关处理事宜。主要包含进程释放、会话（进程组）终止和程序退出处理函数以及杀死进程、终止进程、挂起进程等系统调用函数。还包括进程信号发送函数 `send_sig()` 和通知父进程子进程终止的函数 `tell_father()`。

释放进程的函数 `release()` 主要根据指定的任务数据结构（任务描述符）指针，在任务数组中删除指定的进程指针、释放相关内存页，并立刻让内核重新调度任务的运行。

进程组终止函数 `kill_session()` 通过向会话号与当前进程相同的进程发送挂断进程的信号。

系统调用 `sys_kill()` 用于向进程发送任何指定的信号。根据参数 `pid`（进程标识号）不同的数值，该系统调用会向不同的进程或进程组发送信号。程序注释中已经列出了各种不同情况的处理方式。

程序退出处理函数 `do_exit()` 是在 `exit` 系统调用的中断处理程序中被调用。它首先会释放当前进程的代码段和数据段所占的内存页面。如果当前进程有子进程，就将子进程的 `father` 置为 1，即把子进程的父进程改为进程 1（init 进程）。如果该子进程已经处于僵死状态，则向进程 1 发送子进程终止信号 `SIGCHLD`。接着关闭当前进程打开的所有文件、释放使用的终端设备、协处理器设备，若当前进程是进程组的领头进程，则还需要终止所有相关进程。随后把当前进程置为僵死状态，设置退出码，并向其父进程发送子进程终止信号 `SIGCHLD`。最后让内核重新调度任务的运行。

系统调用 `waitpid()` 用于挂起当前进程，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。该函数的具体操作也要根据其参数进行不同的处理。详见代码中的相关注释。

8.9.2 代码注释

程序 8-8 linux/kernel/exit.c

```

1  /*
2  *  linux/kernel/exit.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。(Linus 从 minix 中引进的)
8  #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。
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
11 #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。
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 <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。
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
16 int sys_pause(void);        // 把进程置为睡眠状态，直到收到信号（kernel/sched.c，144 行）。
17 int sys_close(int fd);      // 关闭指定文件的系统调用（fs/open.c，192 行）。
18

```



```

19 void release(struct task\_struct * p)
20 {
21     int i;
22
23     if (!p)                // 如果进程数据结构指针是 NULL，则什么也不做，退出。
24         return;
25     for (i=1 ; i<NR\_TASKS ; i++)    // 扫描任务数组，寻找指定任务。
26         if (task[i]==p) {
27             task[i]=NULL;           // 置空该任务项并释放相关内存页。
28             free\_page((long)p);
29             schedule();           // 重新调度（似乎没有必要）。
30             return;
31         }
32     panic("trying to release non-existent task"); // 指定任务若不存在则死机。
33 }
34
35 // 向指定任务 p 发送信号 sig，权限为 priv。
36 // 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
37 // 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。
38 // 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。
39 static inline int send\_sig(long sig, struct task\_struct * p, int priv)
40 {
41     // 若信号不正确或任务指针为空则出错退出。
42     if (!p || sig<1 || sig>32)
43         return -EINVAL;
44     // 如果强制发送标志置位，或者当前进程的有效用户标识符(euid)就是指定进程的 euid(也即是自己)，
45     // 或者当前进程是超级用户，则向进程 p 发送信号 sig，即在进程 p 位图中添加该信号，否则出错退出。
46     // 其中 suser() 定义为(current->euid==0)，用于判断是否是超级用户。
47     if (priv || (current->euid==p->euid) || suser())
48         p->signal |= (1<<(sig-1));
49     else
50         return -EPERM;
51     return 0;
52 }
53
54 // 终止会话(session)。
55 // 进程会话的概念请参见第 7 章中有关进程组和会话的说明。
56 static void kill\_session(void)
57 {
58     struct task\_struct **p = NR\_TASKS + task;    // 指针*p 首先指向任务数组最末端。
59
60     // 扫描任务指针数组，对于所有的任务（除任务 0 以外），如果其会话号 session 等于当前进程的
61     // 会话号就向它发送挂断进程信号 SIGHUP。
62     while (--p > &FIRST\_TASK) {
63         if (*p && (*p)->session == current->session)
64             (*p)->signal |= 1<<(SIGHUP-1); // 发送挂断进程信号。
65     }
66 }

```

```

55
56 /*
57  * XXX need to check permissions needed to send signals to process
58  * groups, etc. etc. kill() permissions semantics are tricky!
59  */
60 /*
61  * 为了向进程组等发送信号，XXX 需要检查许可。kill() 的许可机制非常巧妙!
62  * /
63 // 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©。
64 // 参数 pid 是进程号；sig 是需要发送的信号。
65 // 如果 pid 值>0，则信号被发送给进程号是 pid 的进程。
66 // 如果 pid=0，那么信号就会被发送给当前进程的进程组中的所有进程。
67 // 如果 pid=-1，则信号 sig 就会发送给除第一个进程（初始进程 init）外的所有进程。
68 // 如果 pid < -1，则信号 sig 将发送给进程组-pid 的所有进程。
69 // 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。
70 // 该函数扫描任务数组表，并根据 pid 的值对满足条件的进程发送指定的信号 sig。若 pid 等于 0，
71 // 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。
72 int sys_kill(int pid, int sig)
73 {
74     struct task_struct **p = NR_TASKS + task;
75     int err, retval = 0;
76
77     if (!pid) while (--p > &FIRST_TASK) {
78         if (*p && (*p)->pgrp == current->pid)
79             if (err=send_sig(sig, *p, 1)) // 强制发送信号。
80                 retval = err;
81     } else if (pid>0) while (--p > &FIRST_TASK) {
82         if (*p && (*p)->pid == pid)
83             if (err=send_sig(sig, *p, 0))
84                 retval = err;
85     } else if (pid == -1) while (--p > &FIRST_TASK)
86         if (err = send_sig(sig, *p, 0))
87             retval = err;
88     else while (--p > &FIRST_TASK)
89         if (*p && (*p)->pgrp == -pid)
90             if (err = send_sig(sig, *p, 0))
91                 retval = err;
92     return retval;
93 }
94
95 // 通知父进程 -- 向进程 pid 发送信号 SIGCHLD：默认情况下子进程将停止或终止。
96 // 如果没有找到父进程，则自己释放。但根据 POSIX.1 要求，若父进程已先行终止，则子进程应该
97 // 被初始进程 1 收容。
98 static void tell_father(int pid)
99 {
100     int i;
101
102     if (pid)
103         // 扫描进程数组表，寻找指定进程 pid，并向其发送子进程将停止或终止信号 SIGCHLD。
104         for (i=0; i<NR_TASKS; i++) {
105             if (!task[i])
106                 continue;
107             if (task[i]->pid != pid)

```



```

92         continue;
93         task[i]->signal |= (1<<(SIGCHLD-1));
94         return;
95     }
96     /* if we don't find any fathers, we just release ourselves */
97     /* This is not really OK. Must change it to make father 1 */
98     /* 如果没有找到父进程，则进程就自己释放。这样做并不好，必须改成由进程 1 充当其父进程。*/
99     printk("BAD BAD - no father found\n\r");
100     release(current); // 如果没有找到父进程，则自己释放。
101 }
102
103 // 程序退出处理函数。在下面 137 行处的系统调用处理函数 sys_exit() 中被调用。
104 // 该函数将把当前进程置为 TASK_ZOMBIE 状态，然后去执行调度函数 schedule()，不再返回。
105 // 参数 code 是退出状态码，或称为错误码。
106 int do_exit(long code)
107 {
108     int i;
109
110     // 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
111     // (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)
112     // 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
113     // (current->ldt[2] 给出进程数据段描述符的位置)；get_limit() 中的 0x0f 是进程代码段的
114     // 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
115     // 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
116     // 文件的 105 行，get_base() 和 get_limit() 宏位于 include/linux/sched.h 头文件的 213 行处。
117     free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
118     free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
119     // 如果当前进程有子进程，就将子进程的 father 置为 1 (其父进程改为进程 1，即 init 进程)。
120     // 如果该子进程已经处于僵死 (ZOMBIE) 状态，则向进程 1 发送子进程终止信号 SIGCHLD。
121     for (i=0 ; i<NR_TASKS ; i++)
122         if (task[i] && task[i]->father == current->pid) {
123             task[i]->father = 1;
124             if (task[i]->state == TASK_ZOMBIE)
125                 /* assumption task[1] is always init */ /* 这里假设 task[1] 肯定是进程 init */
126                 (void) send_sig(SIGCHLD, task[1], 1);
127         }
128     // 关闭当前进程打开着的所有文件。
129     for (i=0 ; i<NR_OPEN ; i++)
130         if (current->filp[i])
131             sys_close(i);
132     // 对当前进程的工作目录 pwd、根目录 root 以及执行程序文件的 i 节点进行同步操作，放回各个
133     // i 节点并分别置空 (释放)。
134     iput(current->pwd);
135     current->pwd=NULL;
136     iput(current->root);
137     current->root=NULL;
138     iput(current->executable);
139     current->executable=NULL;
140     // 如果当前进程是会话头领 (leader) 进程并且其有控制终端，则释放该终端。
141     if (current->leader && current->tty >= 0)
142         tty_table[current->tty].pgrp = 0;
143     // 如果当前进程上次使用过协处理器，则将 last_task_used_math 置空。
144     if (last_task_used_math == current)

```

```

127         last_task_used_math = NULL;
// 如果当前进程是 leader 进程，则终止该会话的所有相关进程。
128         if (current->leader)
129             kill_session();
// 把当前进程置为僵死状态，表明当前进程已经释放了资源。并保存将由父进程读取的退出码。
130         current->state = TASK_ZOMBIE;
131         current->exit_code = code;
// 通知父进程，也即向父进程发送信号 SIGCHLD -- 子进程将停止或终止。
132         tell_father(current->father);
133         schedule(); // 重新调度进程运行，以让父进程处理僵死进程其他的善后事宜。
// 下面 return 语句仅用于去掉警告信息。因为这个函数不返回，所以若在函数名前加关键字
// volatile，就可以告诉 gcc 编译器本函数不会返回的特殊情况。这样可让 gcc 产生更好一
// 些的代码，并且可以不用再写这条 return 语句也不会产生假警告信息。
134         return (-1); /* just to suppress warnings */ /* 仅用于去掉警告信息 */
135     }
136
// 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--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)。
137 int sys_exit(int error_code)
138 {
139     return do_exit((error_code&0xff)<<8);
140 }
141
// 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。
142 int sys_waitpid(pid_t pid, unsigned long * stat_addr, int options)
143 {
144     int flag, code; // flag 标志用于后面表示所选出的子进程处于就绪或睡眠态。
145     struct task_struct ** p;
146
147     verify_area(stat_addr, 4);
148 repeat:
149     flag=0;
// 从任务数组末端开始扫描所有任务，跳过空项、本进程项以及非当前进程的子进程项。
150     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
151         if (!*p || *p == current)
152             continue;
153         if ((*p)->father != current->pid)
154             continue;
// 此时扫描选择到的进程 p 肯定是当前进程的子进程。

```

```

// 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。
155         if (pid>0) {
156             if ((*p)->pid != pid)
157                 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。
158         } else if (!pid) {
159             if ((*p)->pgrp != current->pgrp)
160                 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。
161         } else if (pid != -1) {
162             if ((*p)->pgrp != -pid)
163                 continue;
164         }
// 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。
165         switch ((*p)->state) {
// 子进程 p 处于停止状态时, 如果此时 WUNTRACED 标志没有置位, 表示程序无须立刻返回,
// 于是继续扫描处理其他进程。如果 WUNTRACED 置位, 则把状态信息 0x7f 放入*stat_addr,
// 并立刻返回子进程号 pid。这里 0x7f 表示的返回状态使 WIFSTOPPED() 宏为真。
// 参见 include/sys/wait.h, 14 行。
166             case TASK\_STOPPED:
167                 if (!(options & WUNTRACED))
168                     continue;
169                 put\_fs\_long(0x7f, stat_addr);
170                 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, 并释放该子进程。最后返回子进程的退出
// 码和 pid。
171             case TASK\_ZOMBIE:
172                 current->cutime += (*p)->utime;
173                 current->cstime += (*p)->stime;
174                 flag = (*p)->pid; // 临时保存子进程 pid。
175                 code = (*p)->exit_code; // 取子进程的退出码。
176                 release(*p); // 释放该子进程。
177                 put\_fs\_long(code, stat_addr); // 置状态信息为退出码值。
178                 return flag; // 返回子进程的 pid。
// 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。
179             default:
180                 flag=1;
181                 continue;
182         }
183     }
// 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。如果此时已设置 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重新执行调度。
// 当又开始执行本进程时, 如果本进程没有收到除 SIGCHLD 以外的信号, 则还是重复处理。
// 否则, 返回出错码 '中断的系统调用' 并退出。针对这个出错号用户程序应该再继续调用本

```

```

// 函数等待子进程。
184     if (flag) {
185         if (options & WNOHANG)           // 若 options = WNOHANG, 则立刻返回。
186             return 0;
187         current->state=TASK_INTERRUPTIBLE; // 置当前进程为可中断等待状态。
188         schedule();                       // 重新调度。
189         if (!(current->signal &= ~(1<<(SIGCHLD-1))))
190             goto repeat;
191         else
192             return -EINTR;                // 返回出错码 (中断的系统调用)。
193     }
// 若没有找到符合要求的子进程, 则返回出错码 (子进程不存在)。
194     return -ECHILD;
195 }
196

```

8.10 fork.c 程序

8.10.1 功能描述

fork()系统调用用于创建子进程。Linux 中所有进程都是进程 0(任务 0)的子进程。该程序是 sys_fork() (在 kernel/system_call.s 中从 208 行开始) 系统调用的辅助处理函数集, 给出了 sys_fork()系统调用中使用的两个 C 语言函数: find_empty_process()和 copy_process()。还包括进程内存区域验证与内存分配函数 verify_area()和 copy_mem()。

copy_process()用于创建并复制进程的代码段和数据段以及环境。在进程复制过程中, 工作主要牵涉到进程数据结构中信息的设置。系统首先为新建进程在主内存区中申请一页内存来存放其任务数据结构信息, 并复制当前进程任务数据结构中的所有内容作为新进程任务数据结构的模板。

随后对已复制的任务数据结构内容进行修改。把当前进程设置为新进程的父进程, 清除信号位图并复位新进程各统计值。接着根据当前进程环境设置新进程任务状态段 (TSS) 中各寄存器的值。由于创建进程时新进程返回值应为 0, 所以需要设置 tss.eax = 0。新建进程内核态堆栈指针 tss.esp0 被设置成新进程任务数据结构所在内存页面的顶端, 而堆栈段 tss.ss0 被设置成内核数据段选择符。tss.ldt 被设置为局部表描述符在 GDT 中的索引值。如果当前进程使用了协处理器, 则还需要把协处理器的完整状态保存到新进程的 tss.i387 结构中。

此后系统设置新任务代码段和数据段的基址和段限长, 并复制当前进程内存分页管理的页目录项和页表项。如果父进程中有文件是打开的, 则子进程中相应的文件也是打开着的, 因此需要将对应文件的打开次数增 1。接着在 GDT 中设置新任务的 TSS 和 LDT 描述符项, 其中基址信息指向新进程任务结构中的 tss 和 ldt。最后再将新任务设置成可运行状态, 并向当前进程返回新进程号。

图 8-13 是内存验证函数 verify_area()中验证内存的起始位置和范围的调整示意图。因为内存写验证函数 write_verify()需要以内存页面为单位 (4096 字节) 进行操作, 因此在调用 write_verify()之前, 需要把验证的起始位置调整为页面起始位置, 同时对验证范围作相应调整。

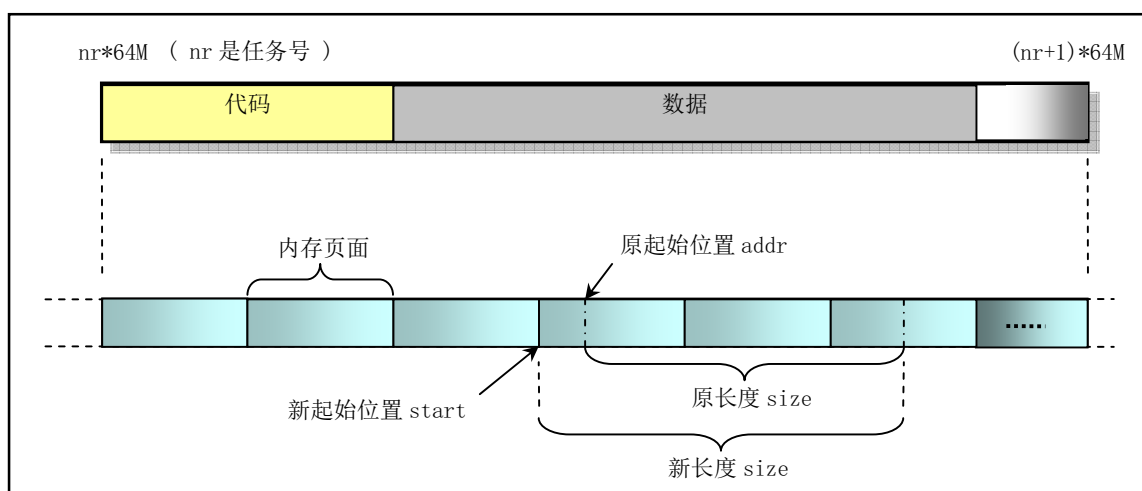


图 8-13 内存验证范围和起始位置的调整

上面根据 fork.c 程序中各函数的功能描述了 fork() 的作用。这里我们从总体上再对其稍加说明。总的来说 fork() 首先会为新进程申请一页内存页用来复制父进程的任务数据结构 (PCB) 信息, 然后会为新进程修改复制的任务数据结构的某些字段值, 包括利用系统调用中断发生时逐步压入堆栈的寄存器信息 (即 copy_process() 的参数) 重新设置任务结构中的 TSS 结构的各字段值, 让新进程的状态保持父进程即将进入中断过程前的状态。然后为新进程确定在线性地址空间中的起始位置 ($nr * 64MB$)。对于 CPU 的分段机制, Linux 0.11 的代码段和数据段在线性地址空间中的位置和长度完全相同。接着系统会为新进程复制父进程的页目录项和页表项。对于 Linux 0.11 内核来说, 所有程序共用一个位于物理内存开始位置处的页目录表, 而新进程的页表则需另行申请一页内存来存放。

在 fork() 的执行过程中, 内核并不会立刻为新进程分配代码和数据内存页。新进程将与父进程共同使用父进程已有的代码和数据内存页面。只有当以后执行过程中如果其中有一个进程以写方式访问内存时被访问的内存页面才会在写操作前被复制到新申请的内存页面中。

8.10.2 代码注释

程序 8-9 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 */
13
14 /*
15 *  '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
16 *  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
17 *  内存管理却有些难度。参见 'mm/memory.c' 中的 'copy_page_tables()' 函数。
18 */

```



```

13 #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)。
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，
                               // 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/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
20 // 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。
21 extern void write_verify(unsigned long address);
22
23 long last_pid=0;             // 最新进程号，其值会由 get_empty_process() 生成。
24
25 // 进程空间区域写前验证函数。
26 // 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
27 // 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。
28 // 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
29 // 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
30 // 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
31 // 到本函数的目的。
32 // 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
33 // 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
34 // 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
35 // 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
36 // 是只读的，则执行共享检验和复制页面操作（写时复制）。
37 void verify_area(void * addr, int size)
38 {
39     unsigned long start;
40
41     // 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。
42     // 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
43     // 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。
44     // 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
45     // 证范围的调整”。
46     start = (unsigned long) addr;
47     size += start & 0xfff;
48     start &= 0xfffff000;           // 此时 start 是当前进程空间中的逻辑地址。
49     // 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
50     // 址位置。对于 Linux 0.11 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。
51     // 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, 261 行）
52     start += get_base(current->ldt[2]);           // include/linux/sched.h, 226 行。
53     while (size>0) {
54         size -= 4096;
55         write_verify(start);
56         start += 4096;
57     }
58 }
59
60 // 复制内存页表。
61 // 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
62 // 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）
63 // 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
64 // 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
44     // 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）。
45     // 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
46     // 空间中的基地址。由于 Linux 0.11 内核还不支持代码和数据段分立的情况，因此这里需要
47     // 检查代码段和数据段基址和限长是否都分别相同。否则内核显示出错信息，并停止运行。
48     // get_limit() 和 get_base() 定义在 include/linux/sched.h 第 226 行处。
49     code_limit=get_limit(0x0f);
50     data_limit=get_limit(0x17);
51     old_code_base = get_base(current->ldt[1]);
52     old_data_base = get_base(current->ldt[2]);
53     if (old_data_base != old_code_base)
54         panic("We don't support separate I&D");
55     if (data_limit < code_limit)
56         panic("Bad data_limit");
57     // 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
58     // 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，
59     // 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。
60     // 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。
61     new_data_base = new_code_base = nr * 0x4000000;
62     p->start_code = new_code_base;
63     set_base(p->ldt[1], new_code_base);
64     set_base(p->ldt[2], new_data_base);
65     if (copy_page_tables(old_data_base, new_data_base, data_limit)) {
66         free_page_tables(new_data_base, data_limit);
67         return -ENOMEM;
68     }
69     return 0;
70 }
71
72 /*
73  * Ok, this is the main fork-routine. It copies the system process
74  * information (task[nr]) and sets up the necessary registers. It
75  * also copies the data segment in it's entirety.
76  */
77 /*
78  * OK, 下面是主要的 fork 子程序。它复制系统进程信息(task[n])
79  * 并且设置必要的寄存器。它还整个地复制数据段。
80  */
81 // 复制进程。
82 // 该函数的参数是进入系统调用中断处理过程（system_call.s）开始，直到调用本系统调用处理
83 // 过程（system_call.s 第 208 行）和调用本函数前时（system_call.s 第 217 行）逐步压入栈的
84 // 各寄存器的值。这些在 system_call.s 程序中逐步压入栈的值（参数）包括：
85 // 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；
86 // ② 第 83--88 行在刚进入 system_call 时压入栈的段寄存器 ds、es、fs 和 edx、ecx、ebx；
87 // ③ 第 94 行调用 sys_call_table 中 sys_fork 函数时压入栈的返回地址（用参数 none 表示）；
88 // ④ 第 212--216 行在调用 copy_process() 之前压入栈的 gs、esi、edi、ebp 和 eax (nr) 值。
89 // 其中参数 nr 是调用 find_empty_process() 分配的任务数组项号。
90 int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,
91                 long ebx, long ecx, long edx,

```

```

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
77     // 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
78     // 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。
79     // 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。
80     p = (struct task\_struct *) get\_free\_page();
81     if (!p)
82         return -EAGAIN;
83     task[nr] = p;
84     *p = *current; /* NOTE! this doesn't copy the supervisor stack */
85                     /* 注意！这样做不会复制超级用户堆栈（只复制进程结构）*/
86     // 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
87     // 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 和父进程
88     // 号 father，并初始化进程运行时间片值等于其 priority 值（一般为 15 个滴答）。接着
89     // 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
90     // 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。
91     p->state = TASK\_UNINTERRUPTIBLE;
92     p->pid = last\_pid; // 新进程号。也由 find_empty_process() 得到。
93     p->father = current->pid; // 设置父进程号。
94     p->counter = p->priority; // 运行时间片值。
95     p->signal = 0; // 信号位图置 0。
96     p->alarm = 0; // 报警定时值（滴答数）。
97     p->leader = 0; /* process leadership doesn't inherit */
98                     /* 进程的领导权是不能继承的 */
99     p->utime = p->stime = 0; // 用户态时间和核心态运行时间。
100    p->cutime = p->cstime = 0; // 子进程用户态和核心态运行时间。
101    p->start_time = jiffies; // 进程开始运行时间（当前时间滴答数）。
102    // 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
103    // 内存，所以 (PAGE_SIZE + (long) p) 让 esp0 正好指向该页顶端。ss0:esp0 用作程序
104    // 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
105    // 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
106    // 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
107    // 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。
108    p->tss.back_link = 0;
109    p->tss.esp0 = PAGE\_SIZE + (long) p; // 任务内核态栈指针。
110    p->tss.ss0 = 0x10; // 内核态栈的段选择符（与内核数据段相同）。
111    p->tss.eip = eip; // 指令代码指针。
112    p->tss.eflags = eflags; // 标志寄存器。
113    p->tss.eax = 0; // 这是当 fork() 返回时新进程会返回 0 的原因所在。
114    p->tss.ecx = ecx;
115    p->tss.edx = edx;
116    p->tss.ebx = ebx;
117    p->tss.esp = esp;
118    p->tss.ebp = ebp;
119    p->tss.esi = esi;
120    p->tss.edi = edi;
121    p->tss.es = es & 0xffff; // 段寄存器仅 16 位有效。
122    p->tss.cs = cs & 0xffff;

```



```

107     p->tss.ss = ss & 0xffff;
108     p->tss.ds = ds & 0xffff;
109     p->tss.fs = fs & 0xffff;
110     p->tss.gs = gs & 0xffff;
111     p->tss.ldt = LDT(nr);          // 任务局部表描述符的选择符 (LDT 描述符在 GDT 中)。
112     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)。
113     if (last_task_used_math == current)
114         __asm__ ("clts ; fnsave %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。
115     if (copy_mem(nr, p)) {          // 返回不为 0 表示出错。
116         task[nr] = NULL;
117         free_page((long) p);
118         return -EAGAIN;
119     }
// 如果父进程中有文件是打开的, 则将对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。
120     for (i=0; i<NR_OPEN;i++)
121         if (f=p->filp[i])
122             f->f_count++;
123     if (current->pwd)
124         current->pwd->i_count++;
125     if (current->root)
126         current->root->i_count++;
127     if (current->executable)
128         current->executable->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。set_tss_desc() 和 set_ldt_desc() 的定义参见 include/asm/system.h 文件
// 52—66 行代码。“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)'。
// 程序然后把新进程设置成就绪态。另外在任务切换时, 任务寄存器 tr 由 CPU 自动加载。
// 最后返回新进程号。
129     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY, &(p->tss));
130     set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY, &(p->ldt));
131     p->state = TASK_RUNNING;        /* do this last, just in case */
                                      /* 最后才将新任务置成就绪态, 以防万一 */
132     return last_pid;
133 }
134
// 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)。
135 int find_empty_process(void)
136 {
137     int i;
138

```

```

// 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。
139     repeat:
140         if ((++last_pid)<0) last_pid=1;
141         for(i=0 ; i<NR_TASKS ; i++)
142             if (task[i] && task[i]->pid == last_pid) goto repeat;
143     for(i=1 ; i<NR_TASKS ; i++)          // 任务 0 项被排除在外。
144         if (!task[i])
145             return i;
146     return -EAGAIN;
147 }
148

```

8.10.3 其他信息

8.10.3.1 任务状态段 (TSS) 信息

下面图 8-14 是任务状态段 TSS (Task State Segment) 的内容。每个任务的 TSS 被保存在任务数据结构 task_struct 中。对它的说明请参第 4 章。

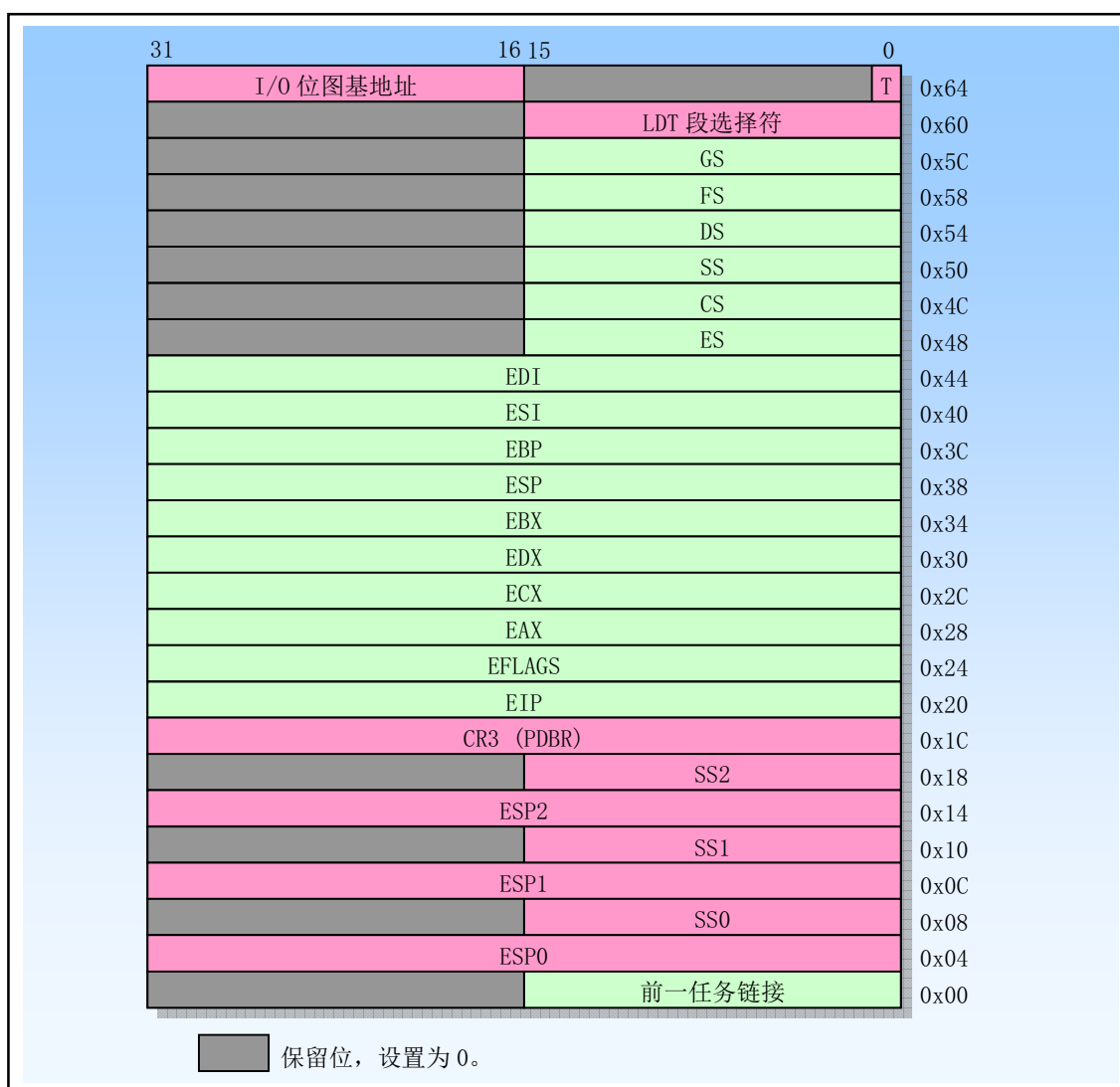


图 8-14 任务状态段 TSS 中的信息。

CPU 管理任务需要的所有信息被存储于一个特殊类型的段中，任务状态段(task state segment - TSS)。图中显示出执行 80386 任务的 TSS 格式。

TSS 中的字段可以分为两类：第 1 类会在 CPU 进行任务切换时动态更新的信息集。这些字段有：通用寄存器（EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI）、段寄存器（ES, CS, SS, DS, FS, GS）、标志寄存器（EFLAGS）、指令指针（EIP）、前一个执行任务的 TSS 的选择符（仅当返回时才更新）。第 2 类字段是 CPU 会读取但不会更改的静态信息集。这些字段有：任务的 LDT 的选择符、含有任务项目录基地址的寄存器（PDBR）、特权级 0-2 的堆栈指针、当任务进行切换时导致 CPU 产生一个调试(debug)异常的 T-比特位（调试跟踪位）、I/O 比特位图基地址（其长度上限就是 TSS 的长度上限，在 TSS 描述符中说明）。

任务状态段可以存放在线形空间的任何地方。与其他各类段相似，任务状态段也是由描述符来定义的。当前正在执行任务的 TSS 是由任务寄存器（TR）来指示的。指令 LTR 和 STR 用来修改和读取任务寄存器中的选择符（任务寄存器的可见部分）。

I/O 比特位图中的每 1 比特对应 1 个 I/O 端口。比如端口 41 的比特位就是 I/O 位图基地址+5，位偏

移 1 处。在保护模式中，当遇到 1 个 I/O 指令时（IN、INS、OUT 和 OUTS），CPU 首先就会检查当前特权级是否小于标志寄存器的 IOPL，如果这个条件满足，就执行该 I/O 操作。如果不满足，那么 CPU 就会检查 TSS 中的 I/O 比特位图。如果相应比特位是置位的，就会产生一般保护性异常，否则就会执行该 I/O 操作。

如果 I/O 位图基址被设置成大于或等于 TSS 段限长，则表示该 TSS 段没有 I/O 许可位图，那么对于所有当前特权层 CPL>IOPL 的 I/O 指令均会导致发生异常保护。在默认情况下，Linux 0.11 内核中把 I/O 位图基址设置成了 0x8000，显然大于 TSS 段限长 104 字节，因此 Linux 0.11 内核中没有 I/O 许可位图。

在 Linux 0.11 中，图中 SS0:ESP0 用于存放任务在内核态运行时的堆栈指针。SS1:ESP1 和 SS2:ESP2 分别对应运行于特权级 1 和 2 时使用的堆栈指针，这两个特权级在 Linux 中没有使用。而任务工作于用户态时堆栈指针则保存在 SS:ESP 寄存器中。由上所述可知，每当任务进入内核态执行时，其内核态堆栈指针初始位置不变，均为任务数据结构所在页面的顶端位置处。

8.11 sys.c 程序

8.11.1 功能描述

sys.c 程序含有很多系统调用功能的实现函数。其中，函数若仅有返回值-ENOSYS，则表示本版 Linux 内核还没有实现该功能，可以参考目前的代码来了解它们的实现方法。所有系统调用功能说明请参见头文件 include/linux/sys.h。

该程序中含有很多有关进程 ID（pid）、进程组 ID（pgrp 或 pgid）、用户 ID（uid）、用户组 ID（gid）、实际用户 ID（ruid）、有效用户 ID（euid）以及会话 ID（session）等的操作函数。下面首先对这些 ID 作一简要说明。

一个用户有用户 ID（uid）和用户组 ID（gid）。这两个 ID 是 passwd 文件中对该用户设置的 ID，通常被称为实际用户 ID（ruid）和实际组 ID（rgid）。而在每个文件的 i 节点信息中都保存着宿主的用户 ID 和组 ID，它们指明了文件拥有者和所属用户组。主要用于访问或执行文件时的权限判别操作。另外，在一个进程的任务数据结构中，为了实现不同功能而保存了 3 种用户 ID 和组 ID。见表 8-5 所示。

表 8-5 与进程相关的用户 ID 和组 ID

类别	用户 ID	组 ID
进程的	uid - 用户 ID。指明拥有该进程的用户。	gid - 组 ID。指明拥有该进程的用户组。
有效的	euid - 有效用户 ID。指明访问文件的权限。	egid - 有效组 ID。指明访问文件的权限。
保存的	suid - 保存的用户 ID。当执行文件的设置用户 ID 标志（set-user-ID）置位时，suid 中保存着执行文件的 uid。否则 suid 等于进程的 euid。	sgid - 保存的组 ID。当执行文件的设置组 ID 标志（set-group-ID）置位时，sgid 中保存着执行文件的 gid。否则 sgid 等于进程的 egid。

进程的 uid 和 gid 分别就是进程拥有者的用户 ID 和组 ID，也即进程的实际用户 ID（ruid）和实际组 ID（rgid）。超级用户可以使用函数 set_uid() 和 set_gid() 对它们进行修改。有效用户 ID 和有效组 ID 用于进程访问文件时的许可权判断。

保存的用户 ID（suid）和保存的组 ID（sgid）用于进程访问设置了 set-user-ID 或 set-group-ID 标志的文件。当执行一个程序时，进程的 euid 通常就是实际用户 ID，egid 通常就是实际组 ID。因此进程只能访问进程的有效用户、有效用户组规定的文件或其他允许访问的文件。但是如果一个文件的 set-user-ID 标志置位时，那么进程的有效用户 ID 就会被设置成该文件宿主的用户 ID，因此进程就可以访问设置了这种标志的受限文件，同时该文件宿主的用户 ID 被保存在 suid 中。同理，文件的 set-group-ID 标志也有

类似的作用并作相同的处理。

例如，如果一个程序的宿主是超级用户，但该程序设置了 `set-user-ID` 标志，那么当该程序被一个进程运行时，则该进程的有效用户 ID (`euclid`) 就会被设置成超级用户的 ID (0)。于是这个进程就拥有了超级用户的权限。一个实际例子就是 Linux 系统的 `passwd` 命令。该命令是一个设置了 `set-user-Id` 的程序，因此允许用户修改自己的口令。因为该程序需要把用户的新口令写入 `/etc/passwd` 文件中，而该文件只有超级用户才有写权限，因此 `passwd` 程序就需要使用 `set-user-ID` 标志。

另外，进程也有标识自己属性的进程 ID (`pid`)、所属进程组的进程组 ID (`pgrp` 或 `pgid`) 和所属会话的会话 ID (`session`)。这 3 个 ID 用于表明进程与进程之间的关系，与用户 ID 和组 ID 无关。

8.11.2 代码注释

程序 8-10 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。(Linus 从 minix 中引进的)。
8
9  #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。
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 <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 #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。
14 #include <sys/utsname.h>    // 系统名称结构头文件。
15
16 // 返回日期和时间。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。
17 int sys_ftime()             // ftime - Fetch time.
18 {
19     return -ENOSYS;
20 }
21
22 //
23 int sys_break()
24 {
25     return -ENOSYS;
26 }
27
28 // 用于当前进程对子进程进行调试(debugging)。
29 int sys_ptrace()
30 {
31     return -ENOSYS;
32 }
33
34 // 改变并打印终端行设置。
35 int sys_stty()
36 {
37     return -ENOSYS;
38 }
39

```

```

// 取终端行设置信息。
36 int sys_gtty()
37 {
38     return -ENOSYS;
39 }
40
// 修改文件名。
41 int sys_rename()
42 {
43     return -ENOSYS;
44 }
45
//
46 int sys_prof()
47 {
48     return -ENOSYS;
49 }
50
// 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 同值。
51 int sys_setregid(int rgid, int egid)
52 {
53     if (rgid>0) {
54         if ((current->gid == rgid) ||
55             suser())
56             current->gid = rgid;
57         else
58             return(-EPERM);
59     }
60     if (egid>0) {
61         if ((current->gid == egid) ||
62             (current->egid == egid) ||
63             (current->sgid == egid) ||
64             suser())
65             current->egid = egid;
66         else
67             return(-EPERM);
68     }
69     return 0;
70 }
71
// 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。
72 int sys_setgid(int gid)
73 {
74     return(sys_setregid(gid, gid));
75 }
76
// 打开或关闭进程计帐功能。
77 int sys_acct()
78 {

```

```

79         return -ENOSYS;
80     }
81     // 映射任意物理内存到进程的虚拟地址空间。
82 int sys_phys()
83 {
84     return -ENOSYS;
85 }
86
87 int sys_lock()
88 {
89     return -ENOSYS;
90 }
91
92 int sys_mpx()
93 {
94     return -ENOSYS;
95 }
96
97 int sys_ulimit()
98 {
99     return -ENOSYS;
100 }
101
102 // 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,
103 // 则时间值也存储在那里。
104 // 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
105 // 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
106 // 函数就可利用 fs 来访问用户空间中的值。
107 int sys_time(long * tloc)
108 {
109     int i;
110
111     i = CURRENT_TIME;
112     if (tloc) {
113         verify_area(tloc, 4);          // 验证内存容量是否够（这里是 4 字节）。
114         put_fs_long(i, (unsigned long *)tloc); // 也放入用户数据段 tloc 处。
115     }
116     return i;
117 }
118
119 /*
120  * Unprivileged users may change the real user id to the effective uid
121  * or vice versa.
122  */
123 /*
124  * 无特权的用户可以见实际的用户标识符（real uid）改成有效的用户标识符
125  *（effective uid），反之亦然。
126  */
127 // 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
128 // 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
129 // 际的用户 ID。保留的 uid（saved uid）被设置成与有效 uid 同值。
130 int sys_setreuid(int ruid, int euid)

```

```

119 {
120     int old_ruid = current->ruid;
121
122     if (ruid>0) {
123         if ((current->euid==ruid) ||
124             (old_ruid == ruid) ||
125             suser())
126             current->ruid = ruid;
127         else
128             return(-EPERM);
129     }
130     if (euid>0) {
131         if ((old_ruid == euid) ||
132             (current->euid == euid) ||
133             suser())
134             current->euid = euid;
135         else {
136             current->ruid = old_ruid;
137             return(-EPERM);
138         }
139     }
140     return 0;
141 }
142
// 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。
143 int sys\_setuid(int uid)
144 {
145     return(sys\_setreuid(uid, uid));
146 }
147
// 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。
148 int sys\_stime(long * tptr)
149 {
150     if (!suser())                // 如果不是超级用户则出错返回（许可）。
151         return -EPERM;
152     startup\_time = get\_fs\_long((unsigned long *)tptr) - jiffies/HZ;
153     return 0;
154 }
155
// 获取当前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。
156 int sys\_times(struct tms * tbuf)
157 {
158     if (tbuf) {
159         verify\_area(tbuf, sizeof *tbuf);

```



```

160         put_fs_long(current->utime, (unsigned long *)&tbuf->tms_utime);
161         put_fs_long(current->stime, (unsigned long *)&tbuf->tms_stime);
162         put_fs_long(current->cutime, (unsigned long *)&tbuf->tms_cutime);
163         put_fs_long(current->cstime, (unsigned long *)&tbuf->tms_cstime);
164     }
165     return jiffies;
166 }
167
// 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。
168 int sys_brk(unsigned long end_data_seg)
169 {
170     // 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。
171     if (end_data_seg >= current->end_code &&
172         end_data_seg < current->start_stack - 16384)
173         current->brk = end_data_seg;
174     return current->brk; // 返回进程当前的数据段结尾值。
175 }
176 /*
177  * This needs some heave checking ...
178  * I just haven't get the stomach for it. I also don't fully
179  * understand sessions/pgrp etc. Let somebody who does explain it.
180  */
181 /*
182  * 下面代码需要某些严格的检查...
183  * 我只是没有胃口来做这些。我也不完全明白 sessions/pgrp 等。还是让了解它们的人来做吧。
184  */
185 // 设置指定进程 pid 的进程组号为 pgid。
186 // 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
187 // 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
188 // 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
189 // 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
190 // 程的相同(193 行)。
191 int sys_setpgid(int pid, int pgid)
192 {
193     int i;
194
195     // 如果参数 pid=0, 则使用当前进程号。如果 pgid 为 0, 则使用当前进程 pid 作为 pgid。
196     // [?? 这里与 POSIX 标准的描述有出入]。
197     if (!pid)
198         pid = current->pid;
199     if (!pgid)
200         pgid = current->pid;
201     // 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, 那么若该
202     // 任务已经是会话首领, 则出错返回。若该任务的会话 ID 与当前进程的不同, 则也出错返回。
203     // 否则设置进程的 pgrp = pgid, 并返回 0。若没有找到指定 pid 的进程, 则返回进程不存在
204     // 出错码。
205     for (i=0 ; i<NR_TASKS ; i++)
206         if (task[i] && task[i]->pid==pid) {
207             if (task[i]->leader)

```

```

192         return -EPERM;
193         if (task[i]->session != current->session)
194             return -EPERM;
195         task[i]->pgrp = pgid;
196         return 0;
197     }
198     return -ESRCH;
199 }
200
201 // 返回当前进程的进程组号。与 getpgid(0) 等同。
202 int sys\_getpgrp(void)
203 {
204     return current->pgrp;
205 }
206
207 // 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。
208 // setsid -- SET Session ID.
209 int sys\_setsid(void)
210 {
211     // 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
212     // 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,
213     // 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。
214     if (current->leader && !suser())
215         return -EPERM;
216     current->leader = 1;
217     current->session = current->pgrp = current->pid;
218     current->tty = -1; // 表示当前进程没有控制终端。
219     return current->pgrp;
220 }
221
222 // 获取系统名称等信息。其中 utsname 结构包含 5 个字段, 分别是: 当前运行系统的名称、网络
223 // 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。
224 // 该结构定义在 include/sys/utsname.h 文件中。
225 int sys\_uname(struct utsname * name)
226 {
227     static struct utsname thisname = { // 这里给出了结构中的信息, 这种编码肯定会改变。
228         "linux.0", "nodename", "release", "version", "machine"
229     };
230     int i;
231
232     // 首先判断参数的有效性。如果存放信息的缓冲区指针为空则出错返回。在验证缓冲区大小是否
233     // 超限 (若超出则内核自动扩展)。然后将 utsname 中的信息逐字节复制到用户缓冲区中。
234     if (!name) return -ERROR;
235     verify\_area(name, sizeof *name);
236     for(i=0; i<sizeof *name; i++)
237         put\_fs\_byte((char *) &thisname[i], i+(char *) name);
238     return 0;
239 }
240
241 // 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。
242 int sys\_umask(int mask)
243 {
244     int old = current->umask;

```

```

233
234     current->umask = mask & 0777;
235     return (old);
236 }
237

```

8.12 vsprintf.c 程序

8.12.1 功能描述

该程序主要包括 vsprintf() 函数，用于对参数产生格式化的输出。由于该函数是 C 函数库中的标准函数，基本没有涉及内核工作原理方面的内容，因此可以跳过。直接阅读代码后对该函数的使用说明。vsprintf() 函数的使用方法请参照 C 库函数手册。

8.12.2 代码注释

程序 8-11 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  */
11  // 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
12  // 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
13  // 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
14  // 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
15  // 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/
16
17  #include <stdarg.h>          // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
18                               // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
19                               // vsprintf、vprintf、vfprintf 函数。
20  #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。
21
22  /* we use this so that we can do without the ctype library */
23  /* 我们使用下面的定义，这样我们就可以不使用 ctype 库了 */
24  #define is_digit(c)        ((c) >= '0' && (c) <= '9') // 判断字符 c 是否为数字字符。
25
26  // 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。
27  static int skip_atoi(const char **s)
28  {
29      int i=0;
30
31      while (is_digit(**s))
32          i = i*10 + *((*s)++) - '0';
33      return i;
34  }

```

```

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: str 字符串指针。
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';
    // 将转数值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; // 返回转换好的字符串。
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，Linux 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 */
103                                     /* 输出字段宽度*/
104     int precision;                  /* min. # of digits for integers; max
105                                     number of chars for from string */
106                                     /* min. 整数数字个数; max. 字符串中字符个数 */
107     int qualifier;                  /* 'h', 'l', or 'L' for integer fields */
108                                     /* 'h', 'l', 或 'L' 用于整数字段 */
109 // 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。
110     for (str=buf; *fmt; ++fmt) {
111         // 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。
112         // 不是格式指示的一般字符均被依次存入 str。
113         if (*fmt != '%') {
114             *str++ = *fmt;
115             continue;
116         }
117         // 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。
118         /* process flags */
119         flags = 0;
120         repeat:
121             ++fmt;                  /* this also skips first '%' */
122             switch (*fmt) {
123                 case '-': flags |= LEFT; goto repeat;    // 左靠齐调整。
124                 case '+': flags |= PLUS; goto repeat;   // 放加号。
125                 case ' ': flags |= SPACE; goto repeat;  // 放空格。
126                 case '#': flags |= SPECIAL; goto repeat; // 是特殊转换。
127                 case '0': flags |= ZEROPAD; goto repeat; // 要填零(即'0')。
128             }
129         // 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。
130         // 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
131         // 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加该标志, 并
132         // 将字段宽度值取为其绝对值。
133         /* get field width */
134         field_width = -1;
135         if (is_digit(*fmt))
136             field_width = skip_atoi(&fmt);
137         else if (*fmt == '*') {
138             /* it's the next argument */ // 这里有个 bug, 应插入 ++fmt;
139             field_width = va_arg(args, int);
140             if (field_width < 0) {
141                 field_width = -field_width;
142                 flags |= LEFT;
143             }
144         }
145         // 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 '.'。
146         // 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
147         // 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
148         // 字段精度值取为 0。
149         /* get the precision */
150         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
152         /* get the conversion qualifier */
153         qualifier = -1;
154         if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
155             qualifier = *fmt;
156             ++fmt;
157         }
158
159         // 下面分析转换指示符。
160         switch (*fmt) {
161             // 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
162             // 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
163             // 参数字符后面添加'宽度值-1'个空格字符。
164             case 'c':
165                 if (!(flags & LEFT))
166                     while (--field_width > 0)
167                         *str++ = ' ';
168                 *str++ = (unsigned char) va_arg(args, int);
169                 while (--field_width > 0)
170                     *str++ = ' ';
171                 break;
172
173             // 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，
174             // 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'
175             // 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
176             // 添加'宽度值-字符串长度'个空格字符。
177             case 's':
178                 s = va_arg(args, char *);
179                 len = strlen(s);
180                 if (precision < 0)
181                     precision = len;
182                 else if (len > precision)
183                     len = precision;
184
185                 if (!(flags & LEFT))
186                     while (len < field_width--)
187                         *str++ = ' ';
188                 for (i = 0; i < len; ++i)
189                     *str++ = *s++;
190                 while (len < field_width--)
191                     *str++ = ' ';

```



```

184                                     break;
185
186 // 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。
187     case 'o':
188         str = number(str, va_arg(args, unsigned long), 8,
189                     field_width, precision, flags);
189     break;
190
191 // 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
192 // 为8，并且需要添零。然后调用 number() 函数进行处理。
193     case 'p':
194         if (field_width == -1) {
195             field_width = 8;
196             flags |= ZEROPAD;
197         }
198         str = number(str,
199                     (unsigned long) va_arg(args, void *), 16,
200                     field_width, precision, flags);
201     break;
202
203 // 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。
204     case 'x':
205         flags |= SMALL;
206     case 'X':
207         str = number(str, va_arg(args, unsigned long), 16,
208                     field_width, precision, flags);
209     break;
210
211 // 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
212 // 带符号标志。'u'代表无符号整数。
213     case 'd':
214     case 'i':
215         flags |= SIGN;
216     case 'u':
217         str = number(str, va_arg(args, unsigned long), 10,
218                     field_width, precision, flags);
219     break;
220
221 // 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。
222 // 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。
223     case 'n':
224         ip = va_arg(args, int *);
225         *ip = (str - buf);
226     break;
227
228 // 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。
229 // 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
230 // 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。
231     default:
232         if (*fmt != '%')
233             *str++ = '%';
234         if (*fmt)
235             *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

```

8.12.3 其他信息

8.12.3.1 vsprintf()的格式字符串

int [vsprintf](#)(char *[buf](#), const char *fmt, [va_list](#) args)

[vsprintf](#)()函数是 [printf](#)()系列函数之一。这些函数都产生格式化的输出：接受确定输出格式的格式字符串 [fmt](#)，用格式字符串对个数变化的参数进行格式化，产生格式化的输出。

[printf](#) 直接把输出送到标准输出句柄 [stdout](#)。[cprintf](#) 把输出送到控制台。[fprintf](#) 把输出送到文件句柄。[printf](#) 前带'v'字符的(例如 [vfprintf](#))表示参数是从 [va_arg](#) 数组的 [va_list](#) args 中接受。[printf](#) 前面带's'字符则表示把输出送到以 null 结尾的字符串 [buf](#) 中（此时用户应确保 [buf](#) 有足够的空间存放字符串）。下面详细说明格式字符串的使用方法。

1. 格式字符串

[printf](#) 系列函数中的格式字符串用于控制函数转换方式、格式化和输出其参数。对于每个格式，必须有对应的参数，参数过多将被忽略。格式字符串中含有两类成份，一种是将被直接复制到输出中的简单字符；另一种是用于对对应参数进行格式化的转换指示字符串。

2. 格式指示字符串

格式指示串的形式如下：

`%[flags][width][.prec][h|lL][type]`

每一个转换指示串均需要以百分号(%)开始。其中

<code>[flags]</code>	是可选择的标志字符序列；
<code>[width]</code>	是可选择的宽度指示符；
<code>[.prec]</code>	是可选择的精度(precision)指示符；
<code>[h lL]</code>	是可选择的输入长度修饰符；
<code>[type]</code>	是转换类型字符(或称为转换指示符)。

[flags](#) 控制输出对齐方式、数值符号、小数点、尾零、二进制、八进制或十六进制等，参见上面列表 27-33 行的注释。标志字符及其含义如下：

表示需要将相应参数转换为“特殊形式”。对于八进制(o)，则转换后的字符串的首位必须是一个零。对于十六进制(x 或 X)，则转换后的字符串需以'0x'或'0X'开头。对于 e,E,f,F,g 以及 G，则即使没有小数位，转换结果也将总是有一个小数点。对于 g 或 G，后拖的零也不会删除。

0 转换结果应该是附零的。对于 d,i,o,u,x,X,e,E,f,g 和 G，转换结果的左边将用零填空而不是用空格。如果同时出现 0 和-标志，则 0 标志将被忽略。对于数值转换，如果给出了精度域，0 标志也被忽略。

- 转换后的结果在相应字段边界内将作左调整（靠左）。（默认是作右调整--靠右）。n 转换例外，转换结果将在右面填充格。
- ' ' 表示带符号转换产生的一个正数结果前应该留一个空格。
- + 表示在一个符号转换结果之前总需要放置一个符号（+或-）。对于默认情况，只有负数使用负号。

width 指定了输出字符串宽度，即指定了字段的最小宽度值。如果被转换的结果要比指定的宽度小，则在其左边（或者右边，如果给出了左调整标志）需要填充空格或零（由 flags 标志确定）的个数等。除了使用数值来指定宽度域以外，也可以使用 '*' 来指出字段的宽度由下一个整型参数给出。当转换值宽度大于 width 指定的宽度时，在任何情况下小宽度值都不会截断结果。字段宽度会扩充以包含完整结果。

precision 是说明输出数字起码的个数。对于 d, l, o, u, x 和 X 转换，精度值指出了至少出现数字的个数。对于 e, E, f 和 F，该值指出在小数点之后出现的数字的个数。对于 g 或 G，指出最大有效数字个数。对于 s 或 S 转换，精度值说明输出字符串的最大字符数。

长度修饰指示符说明了整型数转换后的输出类型形式。下面叙述中‘整型数转换’代表 d, i, o, u, x 或 X 转换。

- hh 说明后面的整型数转换对应于一个带符号字符或无符号字符参数。
- h 说明后面的整型数转换对应于一个带符号整数或无符号短整数参数。
- l 说明后面的整型数转换对应于一个长整数或无符号长整数参数。
- ll 说明后面的整型数转换对应于一个长长整数或无符号长长整数参数。
- L 说明 e, E, f, F, g 或 G 转换结果对应于一个长双精度参数。

type 是说明接受的输入参数类型和输出的格式。各个转换指示符的含义如下：

d, l 整型参数将被转换为带符号整数。如果有精度(precision)的话，则给出了需要输出的最少数字个数。如果被转换的值数字个数较少，就会在其左边添零。默认的精度值是 1。

o, u, x, X 会将无符号的整数转换为无符号八进制(o)、无符号十进制(u)或者是无符号十六进制(x 或 X)表示方式输出。x 表示要使用小写字母(abcdef)来表示十六进制数，X 表示用大写字母(ABCDEF)表示十六进制数。如果存在精度域的话，说明需要输出的最少数字个数。如果被转换的值数字个数较少，就会在其左边添零。默认的精度值是 1。

e, E 这两个转换字符用于经四舍五入将参数转换成[-]d.ddde±dd 的形式。小数点之后的数字个数等于精度。如果没有精度域，就取默认值 6。如果精度是 0，则没有小数出现。E 表示用大写字母 E 来表示指数。指数部分总是用 2 位数字表示。如果数值为 0，那么指数就是 00。

f, F 这两个转换字符用于经四舍五入将参数转换成[-]ddd.ddd 的形式。小数点之后的数字个数等于精度。如果没有精度域，就取默认值 6。如果精度是 0，则没有小数出现。如果有小数点，那么后面起码会有 1 位数字。

g, G 这两个转换字符将参数转换为 f 或 e 的格式（如果是 G，则是 F 或 E 格式）。精度值指定了整数的个数。如果没有精度域，则其默认值为 6。如果精度为 0，则作为 1 来对待。如果转换时指数小于 -4 或大于等于精度，则采用 e 格式。小数部分后拖的零将被删除。仅当起码有一位小数时才会出现小数点。

- c 参数将被转换成无符号字符并输出转换结果。
- s 要求输入为指向字符串的指针，并且该字符串要以 null 结尾。如果有精度域，则只输出精度所要求的字符个数，并且字符串无须以 null 结尾。
- p 以指针形式输出十六进制数。

n 用于把到目前为止转换输出的字符个数保存到由对应输入指针指定的位置中。不对参数进行转换。

% 输出一个百分号%，不进行转换。也即此时整个转换指示为%%。

8.12.4 与当前版本的区别

由于该文件也属于库函数，所以从 1.2 版内核开始就直接使用库中的函数了。也即删除了该文件。

8.13 printk.c 程序

8.13.1 功能描述

printk()是内核中使用的打印（显示）函数，功能与 C 标准函数库中的 printf()相同。重新编写这么一个函数的原因是在内核代码中不能直接使用专用于用户模式的 fs 段寄存器，而需要首先保存它。

不能直接使用 fs 的原因是由于在实际屏幕显示函数 tty_write()中，需要被显示的信息取自于 fs 段指向的数据段中，即用户程序数据段中。而在 printk()函数中需要显示的信息是在内核数据段中，即在内核代码中执行时 ds 指向的内核数据段中。因此在 printk()函数中需要临时使用一下 fs 段寄存器。

printk()函数首先使用 vsprintf()对参数进行格式化处理，然后在保存了 fs 段寄存器的情况下调用 tty_write()进行信息的打印显示。

8.13.2 代码注释

程序 8-12 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  */
12 /*
13  * 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
14  * 的地方。自己编制一个 printf 并在使用前保存 fs，一切就解决了。
15  */
16 // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个类型(va_list)和三个宏
17 // va_start、va_arg 和 va_end，用于 vsprintf、vprintf、vfprintf 函数。
18 #include <stdarg.h>
19 #include <stddef.h> // 标准定义头文件。定义了 NULL，offsetof(TYPE, MEMBER)。
20 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
21
22 static char buf[1024]; // 显示用临时缓冲区。
23
24 // 函数 vsprintf() 定义在 linux/kernel/vsprintf.c 中 92 行开始处。
25 extern int vsprintf(char * buf, const char * fmt, va_list args);

```

```

// 内核使用的显示函数。
// 只能在内核代码中使用。由于实际调用的显示函数 tty_write() 默认使用的显示数据在段寄
// 存器 fs 所指向的用户数据区中, 因此这里需要暂时保存 fs, 并让 fs 指向内核数据段。在显
// 示完之后再恢复原 fs 段的内容。
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
26     // 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
27     // 输出字符串的长度。再运行参数处理结束函数。
28     va_start(args, fmt);        // 在 (include/stdarg.h, 13)
29     i = vsprintf(buf, fmt, args);
30     va_end(args);
31     __asm__( "push %%fs\n\t"      // 保存 fs。
32             "push %%ds\n\t"
33             "pop %%fs\n\t"       // 令 fs = ds。
34             "pushl %0\n\t"       // 将字符串长度压入堆栈(这三个入栈是调用参数)。
35             "pushl $_buf\n\t"    // 将 buf 的地址压入堆栈。
36             "pushl $0\n\t"       // 将数值 0 压入堆栈。是显示通道号 channel。
37             "call _tty_write\n\t" // 调用 tty_write 函数。(chr_drv/tty_io.c, 290)。
38             "addl $8, %%esp\n\t" // 跳过(丢弃)两个入栈参数(buf, channel)。
39             "popl %0\n\t"       // 弹出字符串长度值, 作为返回值。
40             "pop %%fs"         // 恢复原 fs 寄存器。
41             :: "r" (i): "ax", "cx", "dx"); // 通知编译器, 寄存器 ax, cx, dx 值可能已经改变。
42     return i;                  // 返回字符串长度。

```

8.14 panic.c 程序

8.14.1 功能描述

panic()函数用于显示内核错误信息并使系统进入死循环。在内核程序很多地方, 若内核代码在执行过程中出现严重错误时就会调用该函数。在很多情况下调用 panic()函数是一种简明的处理方法。这种做法很好地遵循了 UNIX “尽量简明” 的原则。

panic 是“惊慌, 恐慌”的意思。在 Douglas Adams 的小说《Hitch hikers Guide to the Galaxy》(《银河徒步旅行者指南》)中, 书中最有名的一句话就是“Don't Panic!”。该系列小说是 linux 骇客最常阅读的一类书籍。

8.14.2 代码注释

程序 8-13 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 (include in 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

```

8.15 本章小结

linux/kernel 目录下的 12 个代码文件给出了内核中最为重要的一些机制的实现，主要包括系统调用、进程调度、进程复制以及进程的终止处理四部分。

第9章 块设备驱动程序(block driver)







操作系统的主要功能之一就是与周边的输入输出设备进行通信，采用统一的接口来控制这些外围设备。操作系统的所有设备可以粗略地分成两种类型：块设备(block device)和字符型设备(character device)。块设备是一种可以以固定大小的数据块为单位进行寻址和访问的设备，例如硬盘设备和软盘设备。字符设备是一种以字符流作为操作对象的设备，不能进行寻址操作。例如打印机设备、网络接口设备和终端设备。为了便于管理和访问，操作系统将这些设备统一地以设备号进行分类。在 Linux 0.11 内核中设备被分成 7 类，即共有 7 个设备号（0 到 6）。每个类型中的设备可再根据子（从、次）设备号来加以进一步区别。表 9-1 中列出了各个设备号的设备类型和相关的设备。从表中可以看出某些设备（内存设备）既可以作为块设备也可以作为字符设备进行访问。本章主要讨论和描述块设备驱动程序的实现原理和方法，关于字符设备的讨论放在下一章中进行。

表 9-1 Linux 0.11 内核中的主设备号

主设备号	类型	说明
0	无	无。
1	块/字符	ram,内存设备（虚拟盘等）。
2	块	fd,软驱设备。
3	块	hd,硬盘设备。
4	字符	ttyx 设备（虚拟或串行终端）。
5	字符	tty 设备。
6	字符	lp 打印机设备。

Linux 0.11 内核主要支持硬盘、软盘和内存虚拟盘三种块设备。由于块设备主要与文件系统和高速缓冲有关，因此在继续阅读本章内容之前最好能够先快速浏览一下文件系统一章的内容。本章所涉及的源代码文件见列表 9-1 所示。

列表 9-1 linux/kernel/blk_drv 目录

文件名	大小	最后修改时间(GMT)	说明
 Makefile	1951 bytes	1991-12-05 19:59:42	make 配置文件
 blk.h	3464 bytes	1991-12-05 19:58:01	块设备专用头文件
 floppy.c	11429 bytes	1991-12-07 00:00:38	软盘驱动程序
 hd.c	7807 bytes	1991-12-05 19:58:17	硬盘驱动程序
 ll_rw_blk.c	3539 bytes	1991-12-04 13:41:42	块设备接口程序
 ramdisk.c	2740 bytes	1991-12-06 03:08:06	虚拟盘驱动程序

本程序代码的功能可分为两类，一类是对应各块设备的驱动程序，这类程序有：

- 1. 硬盘驱动程序 hd.c;
- 2. 软盘驱动程序 floppy.c;

3. 内存虚拟盘驱动程序 ramdisk.c;

另一类只有一个程序,是内核中其他程序访问块设备的接口程序 ll_rw_blk.c。块设备专用头文件 blk.h 为这三种块设备与 ll_rw_blk.c 程序交互提供了一个统一的设置方式和相同的设备请求开始程序。

9.1 总体功能

对硬盘和软盘块设备上数据的读写操作是通过中断处理程序进行的。内核每次读写的数据量以一个逻辑块（1024 字节）为单位，而块设备控制器则是以扇区（512 字节）为单位。在处理过程中，使用了读写请求项等待队列来顺序缓冲一次读写多个逻辑块的操作。

当程序需要读取硬盘上的一个逻辑块时，就会向缓冲区管理程序提出申请，而程序的进程则进入睡眠等待状态。缓冲区管理程序首先在缓冲区中寻找以前是否已经读取过这块数据。如果缓冲区中已经有了，就直接将对应的缓冲区块头指针返回给程序并唤醒该程序进程。若缓冲区中还不存在所要求的数据块，则缓冲管理程序就会调用本章中的低级块读写函数 ll_rw_block()，向相应的块设备驱动程序发出一个读数据块的操作请求。该函数会为此创建一个请求结构项，并插入请求队列中。为了提供读写磁盘的效率，减小磁头移动的距离，在插入请求项时使用了电梯移动算法。

此时，若对应块设备的请求项队列为空，则表明此刻该块设备不忙。于是内核就会立刻向该块设备的控制器发出读数据命令。当块设备的控制器将数据读入到指定的缓冲块中后，就会发出中断请求信号，并调用相应的读命令后处理函数，处理继续读扇区操作或者结束本次请求项的过程。例如对相应块设备进行关闭操作和设置该缓冲块数据已经更新标志，最后唤醒等待该块数据的进程。

9.1.1 块设备请求项和请求队列

根据上面描述，我们知道低级读写函数 ll_rw_block()是通过请求项来与各种块设备建立联系并发出读写请求。对于各种块设备，内核使用了一张块设备表 blk_dev[]来进行管理。每种块设备都在块设备表中占有一项。块设备表中每个块设备项的结构为（摘自后面 blk.h）：

```
struct blk_dev_struct {
    void (*request_fn)(void);           // 请求项操作的函数指针。
    struct request * current_request;    // 当前请求项指针。
};
extern struct blk_dev_struct blk_dev[NR_BLK_DEV]; // 块设备表（数组）（NR_BLK_DEV = 7）。
```

其中，第一个字段是一个函数指针，用于操作相应块设备的请求项。例如，对于硬盘驱动程序，它是 do_hd_request()，而对于软盘设备，它就是 do_floppy_request()。第二个字段是当前请求项结构指针，用于指明本块设备目前正在处理的请求项，初始化时都被置成 NULL。

块设备表将在内核初始化时，在 init/main.c 程序调用各设备的初始化函数时被设置。为了便于扩展，Linus 把块设备表建成了一个以主设备号为索引的数组。在 Linux 0.11 中，主设备号有 7 种，见表 9-2 所示。其中，主设备号 1、2 和 3 分别对应块设备：虚拟盘、软盘和硬盘。在块设备数组中其他各项都被默认地置成 NULL。

表 9-2 内核中的主设备号与相关操作函数

主设备号	类型	说明	请求项操作函数
0	无	无。	NULL
1	块/字符	ram, 内存设备（虚拟盘等）。	do_rd_request()
2	块	fd, 软驱设备。	do_fd_request()

3	块	hd,硬盘设备。	do_hd_request()
4	字符	ttyx 设备（虚拟或串行终端等）。	NULL
5	字符	tty 设备。	NULL
6	字符	lp 打印机设备。	NULL

当内核发出一个块设备读写或其他操作请求时, `ll_rw_block()` 函数即会根据其参数中指明的操作命令和数据缓冲块头中的设备号, 利用对应的请求项操作函数 `do_XX_request()` 建立一个块设备请求项 (函数名中的 'XX' 可以是 'rd'、'fd' 或 'hd', 分别代表内存、软盘和硬盘块设备), 并利用电梯算法插入到请求项队列中。请求项队列由请求项数组中的项构成, 共有 32 项, 每个请求项的数据结构如下所示:

```
struct request {
    int dev;                // 使用的设备号 (若为-1, 表示该项空闲)。
    int cmd;                // 命令 (READ 或 WRITE)。
    int errors;             // 操作时产生的错误次数。
    unsigned long sector;   // 起始扇区。(1 块=2 扇区)
    unsigned long nr_sectors; // 读/写扇区数。
    char * buffer;          // 数据缓冲区。
    struct task_struct * waiting; // 任务等待操作执行完成的地方。
    struct buffer_head * bh; // 缓冲区头指针 (include/linux/fs.h, 68)。
    struct request * next;   // 指向下一请求项。
};
extern struct request request[NR_REQUEST]; // 请求项数组 (NR_REQUEST = 32)。
```

每个块设备的当前请求指针与请求项数组中该设备的请求项链表共同构成了该设备的请求队列。项与项之间利用字段 `next` 指针形成链表。因此块设备项和相关的请求队列形成如图 9-1 所示结构。请求项采用数组加链表结构的主要原因是为了满足两个目的: 一是利用请求项的数组结构在搜索空闲请求块时可以进行循环操作, 搜索访问时间复杂度为常数, 因此程序可以编制得很简洁; 二是为满足电梯算法 (Elevator Algorithm) 插入请求项操作, 因此也需要采用链表结构。图 9-1 中示出了硬盘设备当前具有 4 个请求项, 软盘设备具有 1 个请求项, 而虚拟盘设备目前暂时没有读写请求项。

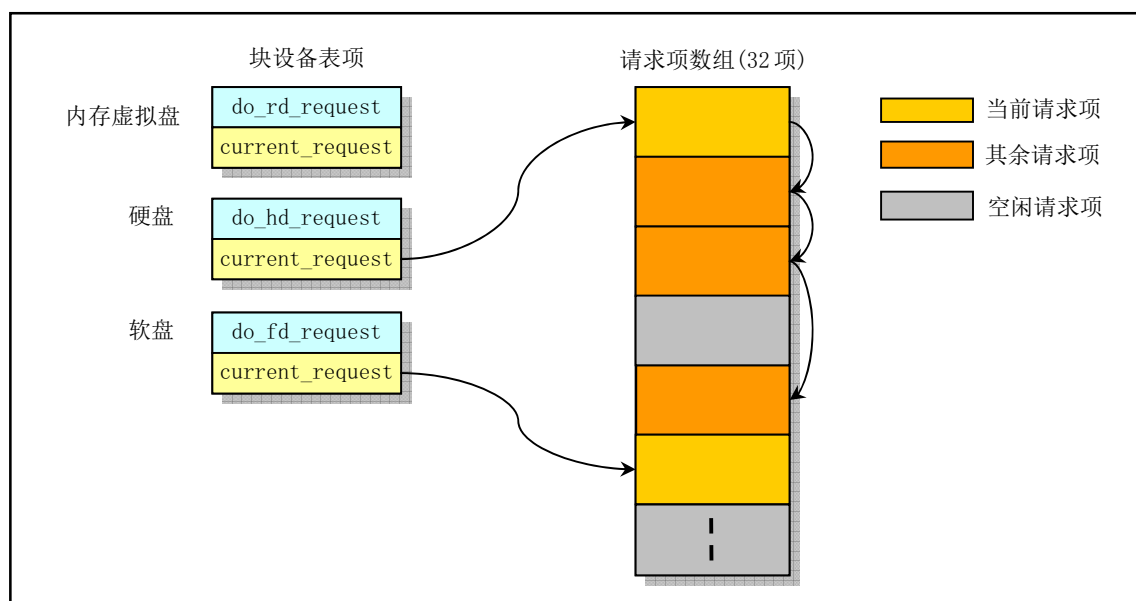


图 9-1 设备表项与请求项

对于一个当前空闲的块设备，当 `ll_rw_block()` 函数为其建立第一个请求项时，会让该设备的当前请求项指针 `current_request` 直接指向刚建立的请求项，并且立刻调用对应设备的请求项操作函数开始执行块设备读写操作。当一个块设备已经有几个请求项组成的链表存在，`ll_rw_block()` 就会利用电梯算法，根据磁头移动距离最小原则，把新建的请求项插入到链表适当的位置处。

另外，为满足读操作的优先权，在为建立新的请求项而搜索请求项数组时，把建立写操作时的空闲项搜索范围限制在整个请求项数组的前 2/3 范围内，而剩下的 1/3 请求项专门给读操作建立请求项使用。

9.1.2 块设备访问调度处理

相对于内存来说，访问硬盘和软盘等块设备中的数据是比较耗时并且影响系统性能的操作。由于硬盘（或软盘）磁头寻道操作（即把读写磁头从一个磁道移动到另一个指定磁道上）需要花费很长时间，因此我们有必要在向硬盘控制器发送访问操作命令之前对读/写磁盘扇区数据的顺序进行排序，即对请求项链表中各请求项的顺序进行排序，使得所有请求项访问的磁盘扇区数据块都尽量依次顺序进行操作。在 Linux 0.1x 内核中，请求项排序操作使用的是电梯算法。其操作原理类似于电梯的运行轨迹 -- 向一个方向移动，直到该方向上最后一个“请求”停止层为止。然后执行反方向移动。对于磁盘来讲就是磁头一直向盘片圆心方向移动，或者反之向盘片边缘移动，参见硬盘结构示意图。

因此，内核并非按照接收到请求项的顺序直接发送给块设备进行处理，而是需要对请求项的顺序进行处理。我们通常把相关的处理程序称为 I/O 调度程序。Linux 0.1x 中的 I/O 调度程序仅对请求项进行了排序处理，而当前流行的 Linux 内核（例如 2.6.x）的 I/O 调度程序中还包含对访问相邻磁盘扇区的两个或多个请求项的合并处理。

9.1.3 块设备操作方式

在系统（内核）与硬盘进行 IO 操作时，需要考虑三个对象之间的交互作用。它们是系统、控制器和驱动器（例如硬盘或软盘驱动器），见图 9-2 所示。系统可以直接向控制器发送命令或等待控制器发出中断请求；控制器在接收到命令后就会控制驱动器的操作，读/写数据或者进行其他操作。因此我们可以把这里控制器发出的中断信号看作是这三者之间的同步操作信号，所经历的操作步骤为：

首先系统指明控制器在执行命令结束而引发的中断过程中应该调用的 C 函数，然后向块设备控制器发送读、写、复位或其他操作命令；

当控制器完成了指定的命令，会发出中断请求信号，引发系统执行块设备的中断处理过程，并在其中调用指定的 C 函数对读/写或其他命令进行命令结束后的处理工作。

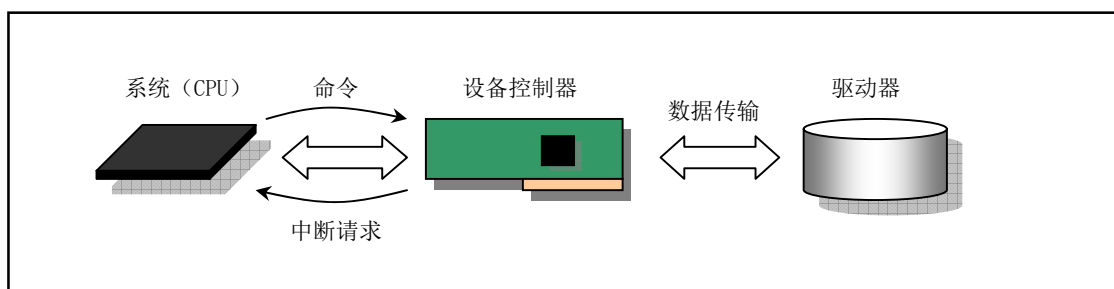


图 9-2 系统、块设备控制器和驱动器

对于写盘操作，系统需要在发出了写命令后（使用 `hd_out()`）等待控制器给予允许向控制器写数据的响应，也即需要查询等待控制器状态寄存器的数据请求服务标志 `DRQ` 置位。一旦 `DRQ` 置位，系统就可以向控制器缓冲区发送一个扇区的数据。

当控制器把数据全部写入驱动器（或发生错误）以后，还会产生中断请求信号，从而在中断处理过程中执行前面预设置的 C 函数（`write_intr()`）。这个函数会查询是否还有数据要写。如果有，系统就再把一个扇区的数据传到控制器缓冲区中，然后再次等待控制器把数据写入驱动器后引发的中断，一直这样重复执行。如果此时所有数据都已经写入驱动器，则该 C 函数就执行本次写盘结束后的处理工作：唤醒等待该请求项有关数据的相关进程、唤醒等待请求项的进程、释放当前请求项并从链表中删除该请求项以及释放锁定的相关缓冲区。最后再调用请求项操作函数去执行下一个读/写盘请求项（若还有的话）。

对于读盘操作，系统在向控制器发送出包括需要读的扇区开始位置、扇区数量等信息的命令后，就等待控制器产生中断信号。当控制器按照读命令的要求，把指定的一扇区数据从驱动器传到了自己的缓冲区之后就会发出中断请求。从而会执行到前面为读盘操作预设置的 C 函数（`read_intr()`）。该函数首先把控制器缓冲区中一个扇区的数据放到系统的缓冲区中，调整系统缓冲区中当前写入位置，然后递减需读的扇区数量。若还有数据要读（递减结果值不为 0），则继续等待控制器发出下一个中断信号。若此时所有要求的扇区都已经读到系统缓冲区中，就执行与上面写盘操作一样的结束处理工作。

对于虚拟盘设备，由于它的读写操作不牵涉到与外部设备之间的同步操作，因此没有上述的中断处理过程。当前请求项对虚拟设备的读写操作完全在 `do_rd_request()` 中实现。

需要提醒的一件事是：在向硬盘或软盘控制器发送了读/写或其他命令后，发送命令函数并不会等待所发命令的执行过程，而是立刻返回调用它的程序中，并最终返回到调用块设备读写函数 `ll_rw_block()` 的其他程序中去等待块设备 IO 的完成。例如高速缓冲区管理程序 `fs/buffer.c` 中的读块函数 `bread()`（第 267 行），在调用了 `ll_rw_block()` 之后，就调用等待函数 `wait_on_buffer()` 让执行当前内核代码的进程立刻进入睡眠状态，直到在相关块设备 IO 结束，在 `end_request()` 函数中被唤醒。

9.2 Makefile 文件

9.2.1 功能描述

该 makefile 文件用于管理对本目录下所有程序的编译。

9.2.2 代码注释

程序 9-1 linux/kernel/blk_drv/Makefile

```

1 #
2 # Makefile for the FREAX-kernel block device drivers.
3 #
4 # Note! Dependencies are done automagically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
8 # FREAX 内核块设备驱动程序的 Makefile 文件
9 # 注意！依赖关系是由 'make dep' 自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
10 # 依赖关系信息放在这里，除非是特别文件的（也即不是一个 .c 文件的信息）。
11 # (Linux 最初的名字叫 FREAX，后来被 ftp.funet.fi 的管理员改成 Linux 这个名字)。
12
13 AR      =gar      # 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。
14 AS      =gas      # GNU 的汇编程序。
15 LD      =gld      # GNU 的连接程序。
16 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
17 CC      =gcc      # GNU C 语言编译器。
18 # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；

```

```

# -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
# 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
# 单短小的函数代码嵌入调用程序中；-mstring-opts Linus 自己添加的优化选项，以后不再使用；
# -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(.././include)。
14 CFLAGS =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
15         -finline-functions -mstring-opts -nostdinc -I../include
# C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
# 出设备或指定的输出文件中；-nostdinc -I../include 同前。
16 CPP     =gcc -E -nostdinc -I../include
17
# 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
# 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
# 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
# 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。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；若目标是.o，而源文件是.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。
18 .c.s:
19     $(CC) $(CFLAGS) \
20     -S -o $.s $<
# 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22     $(AS) -c -o $.o $<
23 .c.o:                                # 类似上面，*.c 文件→*.o 目标文件。不进行连接。
24     $(CC) $(CFLAGS) \
25     -c -o $.o $<
26
27 OBJS = ll_rw_blk.o floppy.o hd.o ramdisk.o      # 定义目标文件变量 OBJS。
28
# 在有了先决条件 OBJS 后使用下面的命令连接成目标 blk_drv.a 库文件。
# 命令行中的 'r' 是操作码和修饰标志（前面可加上 '-'），放置次序任意。其中 'r' 是操作码，
# 说明需要执行的操作。'r' 表示要把命令行末列出的目标文件插入（替换 replacement）归档文件
# blk_drv.a 中。'cs' 是两个修饰标志，用于修饰具体操作行为。'c' 表示当归档文件 blk_drv.a 不
# 存在时就创建这个文件。's' 表示写进或更新归档文件中的目标文件索引。 对一个归档文件单独
# 使用命令 "ar s" 等同于对一个归档文件执行命令 ranlib。
29 blk_drv.a: $(OBJS)
30     $(AR) rcs blk_drv.a $(OBJS)
31     sync
32
# 下面的规则用于清理工作。当执行'make clean'时，就会执行 34--35 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。
33 clean:
34     rm -f core *.o *.a tmp_make
35     for i in *.c;do rm -f `basename $$i .c`.s;done
36
# 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
# 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
# 文件中'### Dependencies' 行后面的所有行（下面从 44 开始的行），并生成 tmp_make
# 临时文件（38 行的作用）。然后对 kernel/blk_drv/目录下的每个 C 文件执行 gcc 预处理操作。
# -M 标志告诉预处理程序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，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
# 文件名加上其依赖关系——该源文件中包含的所有头文件列表。把预处理结果都添加到临时

```



```

# 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
37 dep:
38     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
39     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,`" "; \
40         $(CPP) -M $$i;done) >> tmp_make
41     cp tmp_make Makefile
42
43 ### Dependencies:
44 floppy.s floppy.o : floppy.c ../../include/linux/sched.h ../../include/linux/head.h \
45     ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
46     ../../include/signal.h ../../include/linux/kernel.h \
47     ../../include/linux/fdreg.h ../../include/asm/system.h \
48     ../../include/asm/io.h ../../include/asm/segment.h blk.h
49 hd.s hd.o : hd.c ../../include/linux/config.h ../../include/linux/sched.h \
50     ../../include/linux/head.h ../../include/linux/fs.h \
51     ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
52     ../../include/linux/kernel.h ../../include/linux/hdreg.h \
53     ../../include/asm/system.h ../../include/asm/io.h \
54     ../../include/asm/segment.h blk.h
55 ll_rw_blk.s ll_rw_blk.o : ll_rw_blk.c ../../include/errno.h ../../include/linux/sched.h \
56     ../../include/linux/head.h ../../include/linux/fs.h \
57     ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
58     ../../include/linux/kernel.h ../../include/asm/system.h blk.h

```

9.3 blk.h 文件

9.3.1 功能描述

这是有关硬盘块设备参数的头文件，因为只用于块设备，所以与块设备代码放在同一个地方。其中主要定义了请求等待队列中项的数据结构 **request**，用宏语句定义了电梯搜索算法，并对内核目前支持的虚拟盘，硬盘和软盘三种块设备，根据它们各自的主设备号分别对应了常数值。

关于该文件中定义的“**extern inline**”函数的具体含义，GNU CC 使用手册中的说明如下¹⁰：

如果在函数定义中同时指定了 **inline** 和 **extern** 关键字，则该函数定义仅作为嵌入（内联）使用。并且在任何情况下该函数自身都不会被编译，即使明确地指明其地址也没用。这样的地址只能成为一个外部引用，就好象你仅声明了该函数，而没有定义该函数。

inline 与 **extern** 组合所产生的作用几乎与一个宏（**macro**）相同。使用这种组合的方法是将一个函数定义和这些关键字放在一个头文件中，并且把该函数定义的另一个拷贝（去除 **inline** 和 **extern**）放在一个库文件中。头文件中的函数定义将导致大多数函数调用成为嵌入形式。如果还有其他地方使用该函数，那么它们将引用到库文件中单独的拷贝。

9.3.2 代码注释

程序 9-2 linux/kernel/blk_drv/blk.h

¹⁰ GNU CC 手册 “An Inline Function is As Fast As a Macro”。

```

1 #ifndef BLK_H
2 #define BLK_H
3
4 #define NR_BLK_DEV      7          // 块设备的数量。
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 */
15 /*
16  * 下面定义的 NR_REQUEST 是请求队列中所包含的项数。
17  * 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。
18  *
19  * 32 项好象是一个合理的数字：已经足够从电梯算法中获得好处，
20  * 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
21  * 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）。
22 */
23 #define NR_REQUEST      32
24
25 /*
26  * Ok, this is an expanded form so that we can use the same
27 * request for paging requests when that is implemented. In
28 * paging, 'bh' is NULL, and 'waiting' is used to wait for
29 * read/write completion.
30 */
31 /*
32  * OK, 下面是 request 结构的一个扩展形式，因而当实现以后，我们
33  * 就可以在分页请求中使用同样的 request 结构。在分页处理中，
34  * 'bh' 是 NULL，而 'waiting' 则用于等待读/写的完成。
35 */
36 // 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。
37 // 字段 cmd 可取常量 READ (0) 或 WRITE (1)（定义在 include/linux/fs.h 第 26 行开始处）。
38 struct request {
39     int dev;                /* -1 if no request */ // 发请求的设备号。
40     int cmd;                /* READ or WRITE */ // READ 或 WRITE 命令。
41     int errors;             // 操作时产生的错误次数。
42     unsigned long sector; // 起始扇区。(1 块 = 2 扇区)
43     unsigned long nr_sectors; // 读/写扇区数。
44     char * buffer;          // 数据缓冲区。
45     struct task_struct * waiting; // 任务等待操作执行完成的地方。
46     struct buffer_head * bh; // 缓冲区头指针 (include/linux/fs.h, 68)。
47     struct request * next; // 指向下一请求项。
48 };
49
50 /*
51 * This is used in the elevator algorithm: Note that
52 * reads always go before writes. This is natural: reads
53 * 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：因为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
   // 块设备结构。
45 struct blk_dev_struct {
46     void (*request_fn)(void);           // 请求操作的函数指针。
47     struct request * current_request;    // 当前正在处理的请求信息结构。
48 };
49
50 extern struct blk_dev_struct blk_dev[NR_BLK_DEV]; // 块设备表（数组），每种块设备占用一项。
51 extern struct request request[NR_REQUEST];        // 请求项队列数组。
52 extern struct task_struct * wait_for_request;     // 等待空闲请求项的进程队列头指针。
53
   // 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。
   // 这样下面 61 行—87 行就能为包含本文件的驱动程序给出正确的宏定义。
54 #ifdef MAJOR_NR           // 主设备号。
55
56 /*
57  * Add entries as needed. Currently the only block devices
58  * supported are hard-disks and floppies.
59  */
   /*
   * 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）。
   */
60
61 #if (MAJOR_NR == 1)      // RAM 盘的主设备号是 1。
62 /* ram disk */          /* RAM 盘（内存虚拟盘）*/
63 #define DEVICE_NAME      "ramdisk"           // 设备名称 ramdisk。
64 #define DEVICE_REQUEST do_rd_request        // 设备请求函数 do_rd_request()。
65 #define DEVICE_NR(device) ((device) & 7)    // 设备号（0—7）。
66 #define DEVICE_ON(device)                   // 开启设备。虚拟盘无须开启和关闭。
67 #define DEVICE_OFF(device)                  // 关闭设备。
68
69 #elif (MAJOR_NR == 2)    // 软驱的主设备号是 2。
70 /* floppy */
71 #define DEVICE_NAME      "floppy"           // 设备名称 floppy。
72 #define DEVICE_INTR      do_floppy          // 设备中断处理程序 do_floppy()。
73 #define DEVICE_REQUEST do_fd_request        // 设备请求函数 do_fd_request()。
74 #define DEVICE_NR(device) ((device) & 3)    // 设备号（0—3）。
75 #define DEVICE_ON(device) floppy_on(DEVICE_NR(device)) // 开启设备宏。
76 #define DEVICE_OFF(device) floppy_off(DEVICE_NR(device)) // 关闭设备宏。

```



```

77
78 #elif (MAJOR_NR == 3)    // 硬盘主设备号是 3。
79 /* harddisk */
80 #define DEVICE_NAME "harddisk"           // 硬盘名称 harddisk。
81 #define DEVICE_INTR do_hd                // 设备中断处理程序 do_hd()。
82 #define DEVICE_REQUEST do_hd_request     // 设备请求函数 do_hd_request()。
83 #define DEVICE_NR(device) (MINOR(device)/5) // 设备号 (0—1)。每个硬盘可有 4 个分区。
84 #define DEVICE_ON(device)                // 硬盘一直在工作，无须开启和关闭。
85 #define DEVICE_OFF(device)
86
87 #elif
88 /* unknown blk device */ /* 未知块设备 */
89 #error "unknown blk device"
90
91 #endif
92
93 // 为了便于编程表示，这里定义了两个宏。
94 #define CURRENT (blk_dev[MAJOR_NR].current_request) // 指定主设备号的当前请求项指针。
95 #define CURRENT_DEV DEVICE_NR(CURRENT->dev)         // 当前请求项 CURRENT 中的设备号。
96
97 // 如果定义了设备中断处理函数符号常数 DEVICE_INTR，则把它声明为一个函数指针，并默认为
98 // NULL。例如对于硬盘块设备，前面第 80--86 行宏定义有效，因此下面第 97 行的函数指针定义
99 // 就是“void (*do_hd)(void) = NULL;”。
100 #ifdef DEVICE_INTR
101 void (*DEVICE_INTR)(void) = NULL;
102 #endif
103
104 // 声明符号常数 DEVICE_REQUEST 是一个不带参数并无反回的静态函数指针。
105 static void (DEVICE_REQUEST)(void);
106
107 // 解锁指定的缓冲区（块）。
108 // 如果指定的缓冲区 bh 并没有被上锁，则显示警告信息。否则将该缓冲区解锁，并唤醒等待
109 // 该缓冲区的进程。参数是缓冲区头指针。
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
118 // 结束请求处理。
119 // 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
120 // 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
121 // 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
122 // 项出现的进程，释放并从请求项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。
123 extern inline void end_request(int uptodate)
124 {
125     DEVICE_OFF(CURRENT->dev);           // 关闭设备。
126     if (CURRENT->bh) {                  // CURRENT 为当前请求结构项指针。
127         CURRENT->bh->b_uptodate = uptodate; // 置更新标志。
128         unlock_buffer(CURRENT->bh);       // 解锁缓冲区。
129     }
130     if (!uptodate) {                   // 若更新标志为 0 则显示出错信息。

```

```

117         printk(DEVICE_NAME " I/O error\n|r");
118         printk("dev %04x, block %d\n|r", CURRENT->dev,
119             CURRENT->bh->b_blocknr);
120     }
121     wake_up(&CURRENT->waiting);           // 唤醒等待该请求项的进程。
122     wake_up(&wait_for_request);          // 唤醒等待空闲请求项的进程。
123     CURRENT->dev = -1;                     // 释放该请求项。
124     CURRENT = CURRENT->next;              // 从请求链表中删除该请求项，并且
125 }                                         // 当前请求项指针指向下一个请求项。
126
// 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。于是退出函数。
// 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。
127 #define INIT_REQUEST \
128 repeat: \
129     if (!CURRENT) \                               // 如果当前请求结构指针为 null 则返回。
130         return; \
131     if (MAJOR(CURRENT->dev) != MAJOR_NR) \ // 如果当前设备的主设备号不对则死机。
132         panic(DEVICE_NAME ": request list destroyed"); \
133     if (CURRENT->bh) { \
134         if (!CURRENT->bh->b_lock) \           // 如果请求项的缓冲区没锁定则死机。
135             panic(DEVICE_NAME ": block not locked"); \
136     }
137
138 #endif
139
140 #endif
141

```

9.4 hd.c 程序

9.4.1 功能描述

hd.c 程序是硬盘控制器驱动程序，提供对硬盘控制器块设备的读写驱动和硬盘初始化处理。程序中所有函数按照功能不同可分为 5 类：

- 初始化硬盘和设置硬盘所用数据结构信息的函数，如 sys_setup()和 hd_init()；
- 向硬盘控制器发送命令的函数 hd_out()；
- 处理硬盘当前请求项的函数 do_hd_request()；
- 硬盘中断处理过程中调用的 C 函数，如 read_intr()、write_intr()、bad_rw_intr()和 recal_intr()。do_hd_request()函数也将在 read_intr()和 write_intr()中被调用；
- 硬盘控制器操作辅助函数，如 controler_ready()、drive_busy()、win_result()、hd_out()和 reset_controler()等。

sys_setup()函数利用 boot/setup.s 程序提供的信息对系统中所含硬盘驱动器的参数进行了设置。然后读取硬盘分区表，并尝试把启动引导盘上的虚拟盘根文件系统映像文件复制到内存虚拟盘中，若成功则

加载虚拟盘中的根文件系统，否则就继续执行普通根文件系统加载操作。

`hd_init()`函数用于在内核初始化时设置硬盘控制器中断描述符，并复位硬盘控制器中断屏蔽码，以允许硬盘控制器发送中断请求信号。

`hd_out()`是硬盘控制器操作命令发送函数。该函数带有一个中断过程中调用的 C 函数指针参数，在向控制器发送命令之前，它首先使用这个参数预置好中断过程中会调用的函数指针（`do_hd`，例如 `read_intr()`），然后它按照规定的方式依次向硬盘控制器 0x1f0 至 0x1f7 端口发送命令参数块，随后就立刻退出函数返回而并不会等待硬盘控制器执行读写命令。除控制器诊断（`WIN_DIAGNOSE`）和建立驱动器参数（`WIN_SPECIFY`）两个命令以外，硬盘控制器在接收到任何其他命令并执行了命令以后，都会向 CPU 发出中断请求信号，从而引发系统去执行硬盘中断处理过程（在 `system_call.s`，221 行）。

`do_hd_request()`是硬盘请求项的操作函数。其操作流程如下：

- 首先判断当前请求项是否存在，若当前请求项指针为空，则说明目前硬盘块设备已经没有待处理的请求项，因此立刻退出程序。这是在宏 `INIT_REQUEST` 中执行的语句。否则就继续处理当前请求项。
- 对当前请求项中指明的设备号和请求的盘起始扇区号的合理性进行验证；
- 根据当前请求项提供的信息计算请求数据的磁盘磁道号、磁头号和柱面号；
- 如果复位标志（`reset`）已被设置，则也设置硬盘重新校正标志（`recalibrate`），并对硬盘执行复位操作，向控制器重新发送“建立驱动器参数”命令（`WIN_SPECIFY`）。该命令不会引发硬盘中断；
- 如果重新校正标志被置位的话，就向控制器发送硬盘重新校正命令（`WIN_RESTORE`），并在发送之前预先设置好该命令引发的中断中需要执行的 C 函数（`recal_intr()`），并退出。`recal_intr()`函数的主要作用是：当控制器执行该命令结束并引发中断时，能重新（继续）执行本函数。
- 如果当前请求项指定是写操作，则首先设置硬盘控制器调用的 C 函数为 `write_intr()`，向控制器发送写操作的命令参数块，并循环查询控制器的状态寄存器，以判断请求服务标志（`DRQ`）是否置位。若该标志置位，则表示控制器已“同意”接收数据，于是接着就把请求项所指缓冲区中的数据写入控制器的数据缓冲区中。若循环查询超时后该标志仍然没有置位，则说明此次操作失败。于是调用 `bad_rw_intr()`函数，根据处理当前请求项发生的出错次数来确定是放弃继续当前请求项还是需要设置复位标志，以继续重新处理当前请求项。
- 如果当前请求项是读操作，则设置硬盘控制器调用的 C 函数为 `read_intr()`，并向控制器发送读盘操作命令。

`write_intr()`是在当前请求项是写操作时被设置成中断过程调用的 C 函数。控制器完成写盘命令后会立刻向 CPU 发送中断请求信号，于是在控制器写操作完成后就会立刻调用该函数。

该函数首先调用 `win_result()`函数，读取控制器的状态寄存器，以判断是否有错误发生。若在写盘操作时发生了错误，则调用 `bad_rw_intr()`，根据处理当前请求项发生的出错次数来确定是放弃继续当前请求项还是需要设置复位标志，以继续重新处理当前请求项。若没有发生错误，则根据当前请求项中指明的需写扇区总数，判断是否已经把此请求项要求的所有数据写盘了。若还有数据需要写盘，则使用 `port_write()`函数再把一个扇区的数据复制到控制器缓冲区中。若数据已经全部写盘，则调用 `end_request()`函数来处理当前请求项的结束事宜：唤醒等待本请求项完成的进程、唤醒等待空闲请求项的进程（若有的话）、设置当前请求项所指缓冲区数据已更新标志、释放当前请求项（从块设备链表中删除该项）。最后继续调用 `do_hd_request()`函数，以继续处理硬盘设备的其他请求项。

`read_intr()`则是在当前请求项是读操作时被设置成中断过程中调用的 C 函数。控制器在把指定的扇区数据从硬盘驱动器读入自己的缓冲区后，就会立刻发送中断请求信号。而该函数的主要作用就是把控制器中的数据复制到当前请求项指定的缓冲区中。

与 `write_intr()`开始的处理方式相同，该函数首先也调用 `win_result()`函数，读取控制器的状态寄存器，以判断是否有错误发生。若在读盘时发生了错误，则执行与 `write_intr()`同样的处理过程。若没有发生任何错误，则使用 `port_read()`函数从控制器缓冲区把一个扇区的数据复制到请求项指定的缓冲区中。然后

根据当前请求项中指明的欲读扇区总数，判断是否已经读取了所有的数据。若还有数据要读，则退出，以等待下一个中断的到来。若数据已经全部获得，则调用 `end_request()` 函数来处理当前请求项的结束事宜：唤醒等待当前请求项完成的进程、唤醒等待空闲请求项的进程（若有的话）、设置当前请求项所指缓冲区数据已更新标志、释放当前请求项（从块设备链表中删除该项）。最后继续调用 `do_hd_request()` 函数，以继续处理硬盘设备的其他请求项。

为了能更清晰的看清楚硬盘读写操作的处理过程，我们可以把这些函数、中断处理过程以及硬盘控制器三者之间的执行时序关系用图 9-3 和图 9-4 表示出来。

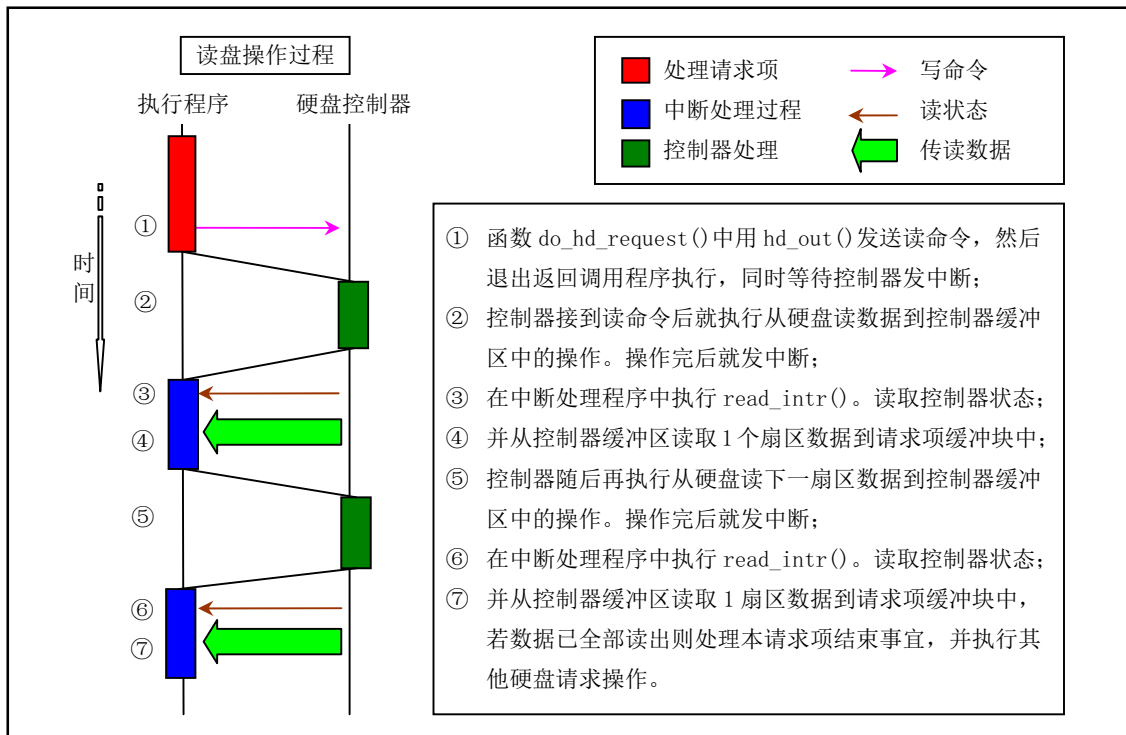


图 9-3 读硬盘数据操作的时序关系

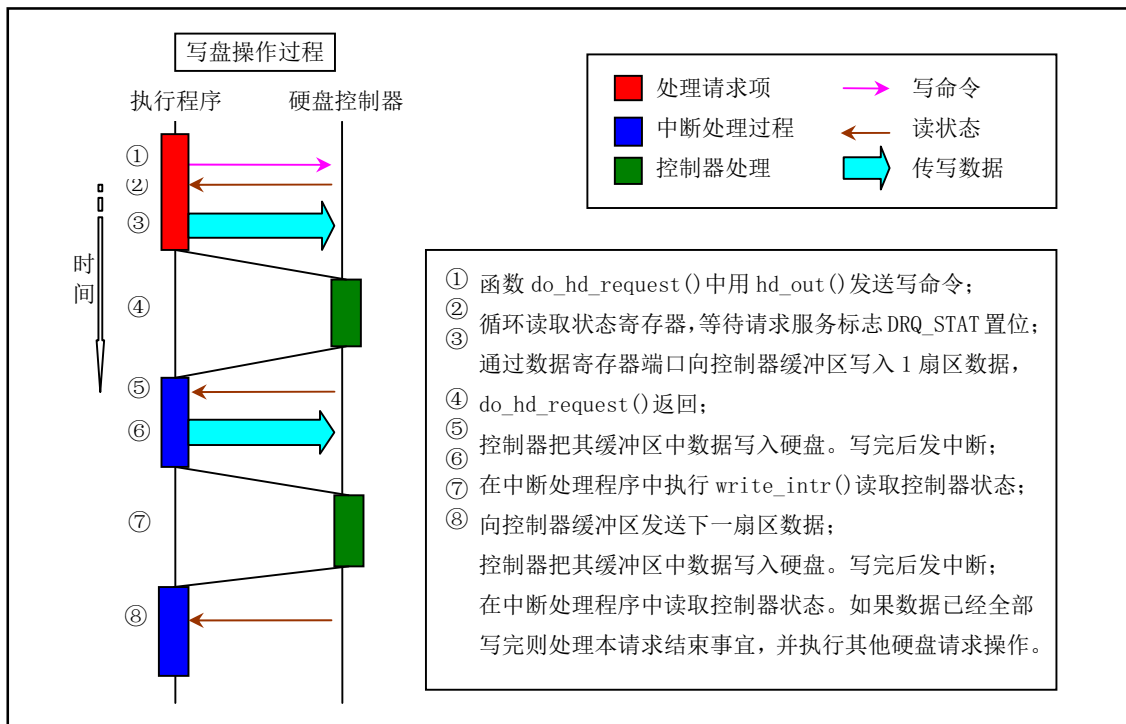


图 9-4 写硬盘数据操作的时序关系

由以上分析可以看出，本程序中最重要的是 4 个函数是 `hd_out()`、`do_hd_request()`、`read_intr()` 和 `write_intr()`。理解了这 4 个函数的作用也就理解了硬盘驱动程序的操作过程③。

值得注意的是，在使用 `hd_out()` 向硬盘控制器发送了读/写或其他命令后，`hd_out()` 函数并不会等待所发命令的执行过程，而是立刻返回调用它的程序中，例如 `do_hd_request()`。而 `do_hd_request()` 函数也立刻返回上一级调用它的函数（`add_request()`），最终返回到调用块设备读写函数 `ll_rw_block()` 的其他程序（例如 `fs/buffer.c` 的 `bread()` 函数）中去等待块设备 IO 的完成。

9.4.2 代码注释

程序 9-3 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 (287 行)。
37 static void recal_intr(void);
38
// 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。
39 static int recalibrate = 1;
// 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。
40 static int reset = 1;
41
42 /*
43 * This struct defines the HD's and their types.
44 */
// 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。
45 struct hd_i_struct {
46 int head, sect, cyl, wpcom, lzone, ctl;
47 };

// 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。
48 #ifdef HD_TYPE
49 struct hd\_i\_struct hd\_info[] = { HD_TYPE }; // 硬盘信息数组。
50 #define NR_HD ((sizeof (hd\_info))/(sizeof (struct hd\_i\_struct))) // 计算硬盘个数。
51 #else
52 struct hd\_i\_struct hd\_info[] = { {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0} };
53 static int NR_HD = 0;
54 #endif
55
// 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。
56 static struct hd\_struct {
57     long start_sect; // 分区在硬盘中的起始物理（绝对）扇区。
58     long nr_sects; // 分区中扇区总数。
59 } hd[5*MAX_HD]={0,0},};
60
// 读端口嵌入汇编宏。读端口 port，共读 nr 字，保存在 buf 中。
61 #define port\_read(port, buf, nr) \
62 __asm__ ("cld;rep;insw"::"d" (port), "D" (buf), "c" (nr): "cx", "di")
63
// 写端口嵌入汇编宏。写端口 port，共写 nr 字，从 buf 中取数据。
64 #define port\_write(port, buf, nr) \
65 __asm__ ("cld;rep;outsw"::"d" (port), "S" (buf), "c" (nr): "cx", "si")
66
67 extern void hd\_interrupt(void); // 硬盘中断过程 (system_call.s, 221 行)。
68 extern void rd\_load(void); // 虚拟盘创建加载函数 (ramdisk.c, 71 行)。
69
70 /* 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。
71 int sys\_setup(void * BIOS)
72 {
73     static int callable = 1; // 限制本函数只能被调用 1 次的标志。
74     int i, drive;
75     unsigned char cmos_disks;
76     struct partition *p;
77     struct buffer\_head *bh;
78
// 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。
79     if (!callable)
80         return -1;
81     callable = 0;
82 #ifndef HD_TYPE // 如果没有定义 HD_TYPE，则读取。
83     for (drive=0 ; drive<2 ; drive++) {

```



```

84         hd_info[drive].cyl = *(unsigned short *) BIOS;        // 柱面数。
85         hd_info[drive].head = *(unsigned char *) (2+BIOS);    // 磁头数。
86         hd_info[drive].wpcom = *(unsigned short *) (5+BIOS);  // 写前预补偿柱面号。
87         hd_info[drive].ctl = *(unsigned char *) (8+BIOS);     // 控制字节。
88         hd_info[drive].lzone = *(unsigned short *) (12+BIOS); // 磁头着陆区柱面号。
89         hd_info[drive].sect = *(unsigned char *) (14+BIOS);   // 每磁道扇区数。
90         BIOS += 16;                                           // 每个硬盘参数表长 16 字节，这里 BIOS 指向下一表。
91     }
    // 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。
92     if (hd_info[1].cyl)
93         NR_HD=2;                                           // 硬盘数置为 2。
94     else
95         NR_HD=1;
96 #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）。
97     for (i=0 ; i<NR_HD ; i++) {
98         hd[i*5].start_sect = 0;                            // 硬盘起始扇区号。
99         hd[i*5].nr_sects = hd_info[i].head*
100             hd_info[i].sect*hd_info[i].cyl; // 硬盘总扇区数。
101     }
102
103     /*
104         We query CMOS about hard disks : it could be that
105         we have a SCSI/ESDI/etc controller that is BIOS
106         compatable with ST-506, and thus showing up in our
107         BIOS table, but not register compatable, and therefore
108         not present in CMOS.
109
110         Furthurmore, we will assume that our ST-506 drives
111         <if any> are the primary drives in the system, and
112         the ones reflected as drive 1 or 2.
113
114         The first drive is stored in the high nibble of CMOS
115         byte 0x12, the second in the low nibble. This will be
116         either a 4 bit drive type or 0xf indicating use byte 0x19
117         for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.
118
119         Needless to say, a non-zero value means we have
120         an AT controller hard disk for that drive.
121
122     */
123     /*

```

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。

```

*/
124 // 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。
125     if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
126         if (cmos_disks & 0x0f)
127             NR_HD = 2;
128         else
129             NR_HD = 1;
130     else
131         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。
132     for (i = NR_HD ; i < 2 ; i++) {
133         hd[i*5].start_sect = 0;
134         hd[i*5].nr_sects = 0;
135     }
// 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。
136     for (drive=0 ; drive<NR_HD ; drive++) {
137         if (! (bh = bread(0x300 + drive*5, 0))) { // 0x300、0x305 是设备号。
138             printk("Unable to read partition table of drive %d\n",
139                 drive);
140             panic("");
141         }
142         if (bh->b_data[510] != 0x55 || (unsigned char)
143             bh->b_data[511] != 0xAA) { // 判断硬盘标志 0xAA55。
144             printk("Bad partition table on drive %d\n", drive);
145             panic("");
146         }
147         p = 0x1BE + (void *)bh->b_data; // 分区表位于第 1 扇区 0x1BE 处。
148         for (i=1; i<5; i++, p++) {
149             hd[i+5*drive].start_sect = p->start_sect;
150             hd[i+5*drive].nr_sects = p->nr_sects;
151         }
152         brelse(bh); // 释放为存放硬盘数据块而申请的缓冲区。

```

```

153     }
    // 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
    // 加载并存放于虚拟盘中, 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。
    // 最后安装根文件系统 (fs/super.c, 第 242 行)。
154     if (NR_HD)
155         printk("Partition table%s ok. \n\r", (NR_HD>1)? "s": "");
156         rd_load(); // 尝试创建并加载虚拟盘。
157         mount_root(); // 安装根文件系统。
158         return (0);
159 }
160
161 // 判断并循环等待硬盘控制器就绪。
162 // 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)
163 // 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
164 // 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
165 // 回到空闲状态, OK!
166 // 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
167 // 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 165 行语句改写成:
168 // “while (--retries && (inb_p(HD_STATUS)&0x80));” 另外, 由于现在的 PC 机速度都很快,
169 // 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!
170 static int controller_ready(void)
171 {
172     int retries=10000;
173     while (--retries && (inb_p(HD_STATUS)&0xc0)!=0x40);
174     return (retries); // 返回等待循环次数。
175 }
176
177 // 检测硬盘执行命令后的状态。(win 表示温切斯特硬盘的缩写)
178 // 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,
179 // 则需要再读错误寄存器 HD_ERROR (0x1f1)。
180 static int win_result(void)
181 {
182     int i=inb_p(HD_STATUS); // 取状态信息。
183     if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
184         == (READY_STAT | SEEK_STAT))
185         return(0); /* ok */
186     if (i&1) i=inb(HD_ERROR); // 若 ERR_STAT 置位, 则读取错误寄存器。
187     return (1);
188 }
189
190 // 向硬盘控制器发送命令块。
191 // 参数: drive - 硬盘号 (0-1); nsect - 读写扇区数; sect - 起始扇区;
192 // head - 磁头号; cyl - 柱面号; cmd - 命令码 (见控制器命令列表);
193 // intr_addr() - 硬盘中断处理程序中将调用的 C 处理函数指针。
194 // 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
195 // 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
196 // kernel/system_call.s 程序第 221 行处。
197 // 第 184 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”);”。
180 static void hd\_out(unsigned int drive,unsigned int nsect,unsigned int sect,
181                 unsigned int head,unsigned int cyl,unsigned int cmd,
182                 void (*intr_addr)(void))
183 {
184     register int port asm(“dx”);    // 定义局部寄存器变量并放在指定寄存器 dx 中。
185
186     // 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
187     // 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
188     // 硬盘控制器出错，也停机。
189     if (drive>1 || head>15)
190         panic(“Trying to write bad sector”);
191     if (!controller\_ready())    //
192         panic(“HD controller not ready”);
193
194     // 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
195     // 第 54—99 行之间，请特别留意其中的第 81 行和 97 行）。然后在向硬盘控制器发送参数和
196     // 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
197     // 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
198     // 0x1f1-0x1f7 发送 7 字节的参数命令块。
199     do_hd = intr_addr;                // do_hd 函数会在中断程序中被调用。
200     outb\_p(hd\_info[drive].ctl,HD\_CMD);    // 向控制寄存器输出控制字节。
201     port=HD\_DATA;                    // 置 dx 为数据寄存器端口(0x1f0)。
202     outb\_p(hd\_info[drive].wpcom>>2,++port); // 参数：写预补偿柱面号(需除 4)。
203     outb\_p(nsect,++port);              // 参数：读/写扇区总数。
204     outb\_p(sect,++port);                // 参数：起始扇区。
205     outb\_p(cyl,++port);                 // 参数：柱面号低 8 位。
206     outb\_p(cyl>>8,++port);              // 参数：柱面号高 8 位。
207     outb\_p(0xA0|(drive<<4)|head,++port); // 参数：驱动器号+磁头号。
208     outb(cmd,++port);                  // 命令：硬盘控制命令。
209 }
210
211 // 等待硬盘就绪。
212 // 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
213 // 就绪，成功返回 0。若经过一段时间仍为忙，则返回 1。
214 static int drive\_busy(void)
215 {
216     unsigned int i;
217
218     // 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。然后再次
219     // 读取主控制器状态字节，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
220     // 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。
221     for (i = 0; i < 10000; i++)
222         if (READY\_STAT == (inb\_p(HD\_STATUS) & (BUSY\_STAT|READY\_STAT)))
223             break;
224     i = inb(HD\_STATUS);                // 再取主控制器状态字节。
225     i &= BUSY\_STAT | READY\_STAT | SEEK\_STAT;
226     if (i == READY\_STAT | SEEK\_STAT)    // 若仅有就绪或寻道结束标志则返回 0。
227         return(0);
228     printk(“HD controller times out\n\r”); // 否则等待超时，显示信息。并返回 1。
229     return(1);
230 }
231
232 // 诊断复位（重新校正）硬盘控制器。

```

```

217 static void reset\_controller(void)
218 {
219     int    i;
220
221     // 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
222     // 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
223     // 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
224     // 于 1 (表示无错误) 则显示硬盘控制器复位失败信息。
221     outb(4, HD\_CMD); // 向控制寄存器端口发送复位控制字节。
222     for(i = 0; i < 100; i++) nop(); // 等待一段时间。
223     outb(hd\_info[0].ctl & 0x0f, HD\_CMD); // 发送正常控制字节 (不禁止重试、重读)。
224     if (drive\_busy())
225         printk("HD-controller still busy\n|r");
226     if ((i = inb(HD\_ERROR)) != 1)
227         printk("HD-controller reset failed: %02x\n|r", i);
228 }
229
230 // 复位硬盘 nr。
231 // 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”，其中
232 // 函数 recal\_intr() 是在硬盘中断处理程序中调用的重新校正处理函数。
230 static void reset\_hd(int nr)
231 {
232     reset\_controller();
233     hd\_out(nr, hd\_info[nr].sect, hd\_info[nr].sect, hd\_info[nr].head-1,
234           hd\_info[nr].cyl, WIN\_SPECIFY, &recal\_intr);
235 }
236
237 // 意外硬盘中断调用函数。
238 // 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
239 // 空时调用该函数。参见 (kernel/system_call.s, 第 241 行)。
237 void unexpected\_hd\_interrupt(void)
238 {
239     printk("Unexpected HD interrupt\n|r");
240 }
241
242 // 读写硬盘失败处理调用函数。
243 // 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，
244 // 而且对应缓冲区更新标志复位，表示数据没有更新。如果读一扇区时的出错次数已经大于
245 // 3 次，则要求执行复位硬盘控制器操作 (设置复位标志)。
242 static void bad\_rw\_intr(void)
243 {
244     if (++CURRENT->errors >= MAX\_ERRORS)
245         end\_request(0);
246     if (CURRENT->errors > MAX\_ERRORS/2)
247         reset = 1;
248 }
249
250 // 读操作中中断调用函数。
251 // 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。
252 // 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
253 // 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)
254 // 后就会执行该函数。
250 static void read\_intr(void)

```

```

251 {
    // 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。
252     if (win_result()) {                // 若控制器忙、读写错或命令执行错，
253         bad_rw_intr();                // 则进行读写硬盘失败处理。
254         do_hd_request();              // 再次请求硬盘作相应(复位)处理。
255         return;
256     }
    // 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：257 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 system_call.s 程序第 237—238 行。
257     port_read(HD_DATA, CURRENT->buffer, 256); // 读数据到请求结构缓冲区。
258     CURRENT->errors = 0;                  // 清出错次数。
259     CURRENT->buffer += 512;               // 调整缓冲区指针，指向新的空区。
260     CURRENT->sector++;                   // 起始扇区号加 1，
261     if (--CURRENT->nr_sectors) {          // 如果所需读出的扇区数还没读完，则再
262         do_hd = &read_intr;             // 置硬盘调用 C 函数指针为 read_intr()。
263         return;
264     }
    // 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。
265     end_request(1);                      // 数据已更新标志置位 (1)。
266     do_hd_request();
267 }
268
///// 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。
269 static void write_intr(void)
270 {
    // 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。
271     if (win_result()) {                // 如果硬盘控制器返回错误信息，
272         bad_rw_intr();                // 则首先进行硬盘读写失败处理。
273         do_hd_request();              // 再次请求硬盘作相应(复位)处理。
274         return;
275     }
    // 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。
276         if (--CURRENT->nr_sectors) {           // 若还有扇区要写，则
277             CURRENT->sector++;                 // 当前请求起始扇区号+1，
278             CURRENT->buffer += 512;             // 调整请求缓冲区指针，
279             do_hd = &write_intr;               // 置函数指针为 write_intr()，
280             port_write(HD_DATA, CURRENT->buffer, 256); // 向数据端口写 256 字。
281             return;
282         }
// 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。
283         end_request(1);                         // 处理请求结束事宜。
284         do_hd_request();                       // 执行其他硬盘请求操作。
285     }
286
///// 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。
287 static void recal_intr(void)
288 {
289     if (win_result())                          // 若返回出错，则调用 bad_rw_intr()。
290         bad_rw_intr();
291     do_hd_request();
292 }
293
///// 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/system_call.s，221 行。
294 void do_hd_request(void)
295 {
296     int i,r;
297     unsigned int block,dev;
298     unsigned int sec,head,cyl;
299     unsigned int nsect;
300
// 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。
301     INIT REQUEST;
302     dev = MINOR(CURRENT->dev);
303     block = CURRENT->sector;           // 请求的起始扇区。
304     if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
305         end_request(0);
306         goto repeat;                   // 该标号在 blk.h 最后面。
307     }
308     block += hd[dev].start_sect;
309     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）。
310     __asm__ ("divl %4": "=a" (block), "=d" (sec): "0" (block), "1" (0),
311             "r" (hd_info[dev].sect));
312     __asm__ ("divl %4": "=a" (cyl), "=d" (head): "0" (block), "1" (0),
313             "r" (hd_info[dev].head));
314     sec++;                             // 对计算所得当前磁道扇区号进行调整。
315     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”。
316     if (reset) {
317         reset = 0;
318         recalibrate = 1;               // 置需重新校正标志。
319         reset_hd(CURRENT_DEV);
320         return;
321     }
// 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。
322     if (recalibrate) {
323         recalibrate = 0;
324         hd_out(dev, hd_info[CURRENT_DEV].sect, 0, 0, 0,
325              WIN RESTORE, &recal_intr);
326         return;
327     }
// 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
// 见本章前面的硬盘操作读/写时序图。
328     if (CURRENT->cmd == WRITE) {
329         hd_out(dev, nsect, sec, head, cyl, WIN_WRITE, &write_intr);
330         for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
331             /* nothing */;
// 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。
332         if (!r) {
333             bad_rw_intr();
334             goto repeat;           // 该标号在 blk.h 文件最后面。
335         }
336         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。
337     } else if (CURRENT->cmd == READ) {
338         hd_out(dev, nsect, sec, head, cyl, WIN_READ, &read_intr);
339     } else
340         panic("unknown hd-command");
341 }
342
// 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/system_call.s, 第 221 行) 是其中断处理过程。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。
343 void hd_init(void)
344 {
345     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;    // do_hd_request().
346     set_intr_gate(0x2E, &hd_interrupt); // 参见 include/asm/system.h 文件。
347     outb_p(inb_p(0x21)&0xfb, 0x21); // 复位接联的主 8259A int2 的屏蔽位。
348     outb(inb_p(0xA1)&0xbf, 0xA1); // 复位硬盘中断请求屏蔽位（在从片上）。
349 }
350

```

9.4.3 其他信息

9.4.3.1 AT 硬盘接口寄存器

AT 硬盘控制器的编程寄存器端口说明见表 9-3 所示。另外请参见 include/linux/hdreg.h 头文件。

表 9-3 AT 硬盘控制器寄存器端口及作用

端口	名称	读操作	写操作
0x1f0	HD_DATA	数据寄存器 -- 扇区数据（读、写、格式化）	
0x1f1	HD_ERROR, HD_PRECOMP	错误寄存器（错误状态）(HD_ERROR)	写前预补偿寄存器 (HD_PRECOMP)
0x1f2	HD_NSECTOR	扇区数寄存器 -- 扇区数（读、写、检验、格式化）	
0x1f3	HD_SECTOR	扇区号寄存器 -- 起始扇区（读、写、检验）	
0x1f4	HD_LCYL	柱面号寄存器 -- 柱面号低字节（读、写、检验、格式化）	
0x1f5	HD_HCYL	柱面号寄存器 -- 柱面号高字节（读、写、检验、格式化）	

0x1f6	HD_CURRENT	驱动器/磁头寄存器 -- 驱动器号/磁头号(101dhhh, d=驱动器号,h=磁头号)	
0x1f7	HD_STATUS,HD_COMMAND	主状态寄存器 (HD_STATUS)	命令寄存器 (HD_COMMAND)
0x3f6	HD_CMD	---	硬盘控制寄存器 (HD_CMD)
0x3f7		数字输入寄存器 (与 1.2M 软盘合用)	---

下面对各端口寄存器进行详细说明。

◆数据寄存器 (HD_DATA, 0x1f0)

这是一对 16 位高速 PIO 数据传输器，用于扇区读、写和磁道格式化操作。CPU 通过该数据寄存器向硬盘写入或从硬盘读出 1 个扇区的数据，也即要使用命令'rep outsw'或'rep insw'重复读/写 cx=256 字。

◆错误寄存器（读）/写前预补偿寄存器（写）(HD_ERROR, 0x1f1)

在读时，该寄存器存放有 8 位的错误状态。但只有当主状态寄存器(HD_STATUS, 0x1f7)的位 0=1 时该寄存器中的数据才有效。执行控制器诊断命令时的含义与其他命令时的不同。见表 9-4 所示。

在写操作时，该寄存器即作为写前预补偿寄存器。它记录写预补偿起始柱面号。对应于与硬盘基本参数表位移 0x05 处的一个字，需除 4 后输出。目前的硬盘大都忽略该参数。

什么是写前补偿？

早期硬盘每个磁道具有固定的扇区数。并且由于每个扇区有固定 512 个字节，因此每个扇区占用的物理磁道长度就会随着越靠近盘片中心就越短小，从而引起磁介质存放数据的能力下降。因此对于硬盘磁头来说就需要采取一定措施以比较高的密度把一个扇区的数据放到比较小的扇区中。所用的常用方法就是写前预补偿（Write Precompensation）技术。在从盘片边缘算起到靠近盘片中心某个磁道（柱面）位置开始，磁头中的写电流会使用某种方法进行一定的调整。

具体调整方法为：磁盘上二进制数据 0、1 的表示是通过磁记录编码方式（例如 FM，MFM 等）进行记录。若相邻记录位两次磁化翻转，则有可能发生磁场重叠。因此此时读出数据时对应的电波形峰值就会漂移。若记录密度提高，则峰值漂移程度就会加剧，有时可能会引起数据位无法分离识别而导致读数据错误。克服这种现象的办法就是使用写前补偿或读后补偿技术。写前补偿是指在向驱动器送入写数据之前，先按照相对于读出时峰值漂移的反方向预先写入脉冲补偿。若读出时信号峰值会向前漂移，则延迟写入该信号；若读出时信号会向后漂移，则提前写入该信号。这样在读出时，峰值的位置就可以接近正常位置。

表 9-4 硬盘控制器错误寄存器

值	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

在写操作时，该寄存器即作为写前预补偿寄存器。它记录写预补偿起始柱面号。对应于与硬盘基本参数表位移 0x05 处的一个字，需除 4 后输出。

◆扇区数寄存器 (HD_NSECTOR, 0x1f2)

该寄存器存放读、写、检验和格式化命令指定的扇区数。当用于多扇区操作时，每完成 1 扇区的操

作该寄存器就自动减 1，直到为 0。若初值为 0，则表示传输最大扇区数 256。

◆扇区号寄存器 (HD_SECTOR, 0x1f3)

该寄存器存放读、写、检验操作命令指定的扇区号。在多扇区操作时，保存的是起始扇区号，而每完成 1 扇区的操作就自动增 1。

◆柱面号寄存器 (HD_LCYL, HD_HCYL, 0x1f4, 0x1f5)

该两个柱面号寄存器分别存放有柱面号的低 8 位和高 2 位。

◆驱动器/磁头寄存器 (HD_CURRENT, 0x1f6)

该寄存器存放有读、写、检验、寻道和格式化命令指定的驱动器和磁头号。其位格式为 101dhhhh。其中 101 表示采用 ECC 校验码和每扇区为 512 字节；d 表示选择的驱动器 (0 或 1)；hhhh 表示选择的磁头。见表 9-5 所示。

表 9-5 驱动器/磁头寄存器含义

位	名称		说明
0	HS0	磁头号位 0	磁头号最低位。
1	HS1	磁头号位 1	
2	HS2	磁头号位 2	
3	HS3	磁头号位 3	磁头号最高位。
4	DRV	驱动器	选择驱动器，0 - 选择驱动器 0； 1 - 选择驱动器 1。
5	Reserved	保留	总是 1。
6	Reserved	保留	总是 0。
7	Reserved	保留	总是 1。

◆主状态寄存器 (读)/命令寄存器 (写) (HD_STATUS/HD_COMMAND, 0x1f7)

在读时，对应一个 8 位主状态寄存器。反映硬盘控制器在执行命令前后的操作状态。各位的含义见表 9-6 所示。

表 9-6 8 位主状态寄存器

位	名称	屏蔽码	说明
0	ERR_STAT	0x01	命令执行错误。当该位置位时说明前一个命令以出错结束。此时出错寄存器和状态寄存器中的比特位含有引起错误的一些信息。
1	INDEX_STAT	0x02	收到索引。当磁盘旋转到索引标志时会设置该位。
2	ECC_STAT	0x04	ECC 校验错。当遇到一个可恢复的数据错误而且已得到纠正，就会设置该位。这种情况不会中断一个多扇区读操作。
3	DRQ_STAT	0x08	数据请求服务。当该位被置位时，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。
4	SEEK_STAT	0x10	驱动器寻道结束。当该位被置位时，表示寻道操作已经完成，磁头已经停在指定的磁道上。当发生错误时，该位并不会改变。只有主机读取了状态寄存器后，该位就会再次表示当前寻道的完成状态。
5	WRERR_STAT	0x20	驱动器故障 (写出错)。当发生错误时，该位并不会改变。只有主机读取了状态寄存器后，该位就会再次表示当前写操作的出错状态。
6	READY_STAT	0x40	驱动器准备好 (就绪)。表示驱动器已经准备好接收命令。当发生错误时，该位并不会改变。只有主机读取了状态寄存器后，该位就会再次表示当前驱动器就绪状态。在开机时，应该复位该比特位，直到驱动器速度达到正常并且能够接

			收命令。
7	BUSY_STAT	0x80	<p>控制器忙碌。当驱动器正在操作由驱动器的控制器设置该位。此时主机不能发送命令块。而对任何命令寄存器的读操作将返回状态寄存器的值。在下列条件下该位会被置位：</p> <p>在机器复位信号 RESET 变负或者设备控制寄存器的 SRST 被设置之后 400 纳秒以内。在机器复位之后要求该位置位状态不能超过 30 秒。</p> <p>主机在向命令寄存器写重新校正、读、读缓冲、初始化驱动器参数以及执行诊断等命令的 400 纳秒以内。</p> <p>在写操作、写缓冲或格式化磁道命令期间传输了 512 字节数据的 5 微秒之内。</p>

当执行写操作时，该端口对应命令寄存器，接受 CPU 发出的硬盘控制命令，共有 8 种命令，见表 9-7 所示。其中最后一列用于说明相应命令结束后控制器所采取的动作（引发中断或者什么也不做）。

表 9-7 AT 硬盘控制器命令列表

命令名称		命令码字节		默认值	命令执行结束形式
		高 4 位	D3 D2 D1 D0		
WIN_RESTORE	驱动器重新校正(复位)	0x1	R R R R	0x10	中断
WIN_READ	读扇区	0x2	0 0 L T	0x20	中断
WIN_WRITE	写扇区	0x3	0 0 L T	0x30	中断
WIN_VERIFY	扇区检验	0x4	0 0 0 T	0x40	中断
WIN_FORMAT	格式化磁道	0x5	0 0 0 0	0x50	中断
WIN_INIT	控制器初始化	0x6	0 0 0 0	0x60	中断
WIN_SEEK	寻道	0x7	R R R R	0x70	中断
WIN_DIAGNOSE	控制器诊断	0x9	0 0 0 0	0x90	中断或空闲
WIN_SPECIFY	建立驱动器参数	0x9	0 0 0 1	0x91	中断

表中命令码字节的低 4 位是附加参数，其含义为：

R 是步进速率。R=0，则步进速率为 35us；R=1 为 0.5ms，以此量递增。程序中默认 R=0。

L 是数据模式。L=0 表示读/写扇区为 512 字节；L=1 表示读/写扇区为 512 加 4 字节的 ECC 码。程序中默认值是 L=0。

T 是重试模式。T=0 表示允许重试；T=1 则禁止重试。程序中取 T=0。

下面分别对这几个命令进行详细说明。

(1) 0x1X -- （WIN_RESTORE），驱动器重新校正（Recalibrate）命令

该命令把读/写磁头从磁盘上任何位置移动到 0 柱面。当接收到该命令时，驱动器会设置 BUSY_STAT 标志并且发出一个 0 柱面寻道指令。然后驱动器等待寻道操作结束，更新状态、复位 BUSY_STAT 标志并且产生一个中断。

(2) 0x20 -- （WIN_READ）可重试读扇区；0x21 -- 无重试读扇区。

读扇区命令可以从指定扇区开始读取 1 到 256 个扇区。若所指定的命令块（见表 9-9）中扇区计数为 0 的话，则表示读取 256 个扇区。当驱动器接受了该命令，将会设立 BUSY_STAT 标志并且开始执行该命令。对于单个扇区的读取操作，若磁头的磁道位置不对，则驱动器会隐含地执行一次寻道操作。一旦磁头在正确的磁道上，驱动器磁头就会定位到磁道地址场中相应的标志域（ID 域）上。

对于无重试读扇区命令，若两个索引脉冲发生之前不能正确读取无错的指定 ID 域，则驱动器就会在错误寄存器中给出 ID 没有找到的错误信息。对于可重试读扇区命令，驱动器则会在读 ID 域碰到问题时重试多次。重试的次数由驱动器厂商设定。

如果驱动器正确地读到了 ID 域，那么它就需要在指定的字节数中识别数据地址标志（Data Address Mark），否则就报告数据地址标志没有找到的错误。一旦磁头找到数据地址标志，驱动器就会把数据域中的数据读入扇区缓冲区中。如果发生错误，驱动器就会设置出错比特位、设置 DRQ_STAT 并且产生一个中断。不管是否发生错误，驱动器总是会在读扇区后设置 DRQ_STAT。在命令完成后，命令块寄存器中将含有最后一个所读扇区的柱面号、磁头号和扇区号。

对于多扇区读操作，每当驱动器准备好向主机发送一个扇区的数据时就会设置 DRQ_STAT、清 BUSY_STAT 标志并且产生一个中断。当扇区数据传输结束，驱动器就会复位 DRQ_STAT 和 BUSY_STAT 标志，但在最后一个扇区传输完成后会设置 BUSY_STAT 标志。在命令结束后命令块寄存器中将含有最后一个所读扇区的柱面号、磁头号和扇区号。

如果在多扇区读操作中发生了一个不可纠正的错误，读操作将在发生错误的扇区处终止。同样，此时命令块寄存器中将含有该出错扇区的柱面号、磁头号和扇区号。不管错误是否可以被纠正，驱动器都会把数据放入扇区缓冲区中。

(3) 0x30 -- (WIN_WRITE) 可重试写扇区；0x31 -- 无重试写扇区。

写扇区命令可以从指定扇区开始写 1 到 256 个扇区。若所指定的命令块（见表 9-9）中扇区计数为 0 的话，则表示要写 256 个扇区。当驱动器接受了该命令，它将设置 DRQ_STAT 并等待扇区缓冲区被添满数据。在开始第一次向扇区缓冲区添入数据时不会产生中断，一旦数据填满驱动器就会复位 DRQ、设置 BUSY_STAT 标志并且开始执行命令。

对于写一个扇区数据的操作，驱动器会在收到命令时设置 DRQ_STAT 并且等待主机填满扇区缓冲区。一旦数据已被传输，驱动器就会设置 BUSY_STAT 并且复位 DRQ_STAT。与读扇区操作一样，若磁头的磁道位置不对，则驱动器会隐含地执行一次寻道操作。一旦磁头在正确的磁道上，驱动器磁头就会定位到磁道地址场中相应的标志域（ID 域）上。

如果 ID 域被正确地读出，则扇区缓冲区中的数据包括 ECC 字节就被写到磁盘上。当驱动器处理过扇区后就会清 BUSY_STAT 标志并且产生一个中断。此时主机就可以读取状态寄存器。在命令结束后，命令块寄存器中将含有最后一个所写扇区的柱面号、磁头号和扇区号。

在多扇区写操作期间，除了对第一个扇区的操作，当驱动器准备好从主机接收一个扇区的数据时就会设置 DRQ_STAT、清 BUSY_STAT 标志并且产生一个中断。一旦一个扇区传输完毕，驱动器就会复位 DRQ 并设置 BUSY 标志。当最后一个扇区被写到磁盘上后，驱动器就会清掉 BUSY_STAT 标志并产生一个中断（此时 DRQ_STAT 已经复位）。在写命令结束后，命令块寄存器中将含有最后一个所写扇区的柱面号、磁头号和扇区号。

如果在多扇区写操作中发生了一个错误，写操作将在发生错误的扇区处终止。同样，此时命令块寄存器中将含有该出错扇区的柱面号、磁头号和扇区号。

(4) 0x40 -- (WIN_VERIFY) 可重试读扇区验证；0x41 -- 无重试读扇区验证。

该命令的执行过程与读扇区操作相同，但是本命令不会导致驱动器去设置 DRQ_STAT，并且不会向主机传输数据。当收到读验证命令时，驱动器就会设置 BUSY_STAT 标志。当指定的扇区被验证过后，驱动器就会复位 BUSY_STAT 标志并且产生一个中断。在命令结束后，命令块寄存器中将含有最后一个所验证扇区的柱面号、磁头号和扇区号。

如果在多扇区验证操作中发生了一个错误，验证操作将在发生错误的扇区处终止。同样，此时命令块寄存器中将含有该出错扇区的柱面号、磁头号和扇区号。

(5) 0x50 -- (WIN_FORMAT) 格式化磁道命令。

扇区计数寄存器中指定了磁道地址。当驱动器接受该命令时，它会设置 DRQ_STAT 比特位，然后等待主机填满扇区缓冲区。当缓冲区满后，驱动器就会清 DRQ_STAT、设置 BUSY_STAT 标志并

且开始命令的执行。

(6) 0x60 -- (WIN_INIT) 控制器初始化。

(7) 0x7X -- (WIN_SEEK) 寻道操作。

寻道操作命令将命令块寄存器中所选择的磁头移动到指定的磁道上。当主机发出一个寻道命令时，驱动器会设置 BUSY 标志并且产生一个中断。在寻道操作结束之前，驱动器在寻道操作完成之前不会设置 SEEK_STAT (DSC - 寻道完成)。在驱动器产生一个中断之前寻道操作可能还没有完成。如果在寻道操作进行中主机又向驱动器发出了一个新命令，那么 BUSY_STAT 将依然处于置位状态，直到寻道结束。然后驱动器才开始执行新的命令。

(8) 0x90 -- (WIN_DIAGNOSE) 驱动器诊断命令。

该命令执行驱动器内部实现的诊断测试过程。驱动器 0 会在收到该命令的 400ns 内设置 BUSY_STAT 比特位。

如果系统中含有第 2 个驱动器，即驱动器 1，那么两个驱动器都会执行诊断操作。驱动器 0 会等待驱动器 1 执行诊断操作 5 秒钟。如果驱动器 1 诊断操作失败，则驱动器 0 就会在自己的诊断状态中附加 0x80。如果主机在读取驱动器 0 的状态时检测到驱动器 1 的诊断操作失败，它就会设置驱动器/磁头寄存器(0x1f6)的驱动器选择比特位(位 4)，然后读取驱动器 1 的状态。如果驱动器 1 通过诊断检测或者驱动器 1 不存在，则驱动器 0 就直接把自己的诊断状态加载到出错寄存器中。

如果驱动器 1 不存在，那么驱动器 0 仅报告自己的诊断结果，并且在复位 BUSY_STAT 比特位后产生一个中断。

(9) 0x91 -- (WIN_SPECIFY) 建立驱动器参数命令。

该命令用于让主机设置多扇区操作时磁头交换和扇区计数循环值。在收到该命令时驱动器会设置 BUSY_STAT 比特位并产生一个中断。该命令仅使用两个寄存器的值。一个是扇区计数寄存器，用于指定扇区数；另一个是驱动器/磁头寄存器，用于指定磁头数-1，而驱动器选择比特位(位 4)则根据具体选择的驱动器来设置。

该命令不会验证所选择的扇区计数值和磁头数。如果这些值无效，驱动器不会报告错误。直到另一个命令使用这些值而导致无效一个访问错误。

◆ 硬盘控制寄存器(写)(HD_CMD, 0x3f6)

该寄存器是只写的。用于存放硬盘控制字节并控制复位操作。其定义与硬盘基本参数表的位移 0x08 处的字节说明相同，见表 9-8 所示。

表 9-8 硬盘控制字节的含义

位移	大小	说明
0x08	字节	控制字节(驱动器步进选择)
		位 0 未用
		位 1 保留(0)(关闭 IRQ)
		位 2 允许复位
		位 3 若磁头数大于 8 则置 1
		位 4 未用(0)
		位 5 若在柱面数+1 处有生产商的坏区图，则置 1
		位 6 禁止 ECC 重试
		位 7 禁止访问重试。

9.4.3.2 AT 硬盘控制器编程

在对硬盘控制器进行操作控制时，需要同时发送参数和命令。其命令格式见表 9-9 所示。首先发送

6 字节的参数，最后发出 1 字节的命令码。不管什么命令均需要完整输出这 7 字节的命令块，依次写入端口 0x1f1 -- 0x1f7。一旦命令块寄存器加载，命令就开始执行。

表 9-9 命令格式

端口	说明
0x1f1	写预补偿起始柱面号
0x1f2	扇区数
0x1f3	起始扇区号
0x1f4	柱面号低字节
0x1f5	柱面号高字节
0x1f6	驱动器号/磁头号
0x1f7	命令码

首先 CPU 向控制寄存器端口(HD_CMD)0x3f6 输出控制字节，建立相应的硬盘控制方式。方式建立后即可按上面顺序发送参数和命令。步骤为：

1. 检测控制器空闲状态：CPU 通过读主状态寄存器，若位 7 (BUSY_STAT) 为 0，表示控制器空闲。若在规定时间内控制器一直处于忙状态，则判为超时出错。参见 hd.c 中第 161 行的 controller_ready() 函数。
2. 检测驱动器是否就绪：CPU 判断主状态寄存器位 6 (READY_STAT) 是否为 1 来看驱动器是否就绪。为 1 则可输出参数和命令。参见 hd.c 中第 202 行的 drive_busy() 函数。
3. 输出命令块：按顺序输出分别向对应端口输出参数和命令。参见 hd.c 中第 180 行开始的 hd_out() 函数。
4. CPU 等待中断产生：命令执行后，由硬盘控制器产生中断请求信号 (IRQ14 -- 对应中断 int46) 或置控制器状态为空闲，表明操作结束或表示请求扇区传输（多扇区读/写）。程序 hd.c 中在中断处理过程中调用的函数参见代码 237--293 行。有 5 个函数分别对应 5 种情况。
5. 检测操作结果：CPU 再次读主状态寄存器，若位 0 等于 0 则表示命令执行成功，否则失败。若失败则可进一步查询错误寄存器(HD_ERROR)取错误码。参见 hd.c 中第 202 行的 win_result() 函数。

9.4.3.3 硬盘基本参数表

中断向量表中，int 0x41 的中断向量位置 (4 * 0x41 = 0x0000:0x0104) 存放的并不是中断程序的地址而是第一个硬盘的基本参数表，见表 9-10 所示。对于 100%兼容的 BIOS 来说，这里存放着硬盘参数表阵列的首地址 F000h:E401h。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向量中。

表 9-10 硬盘基本参数信息表

位移	大小	说明
0x00	字	柱面数
0x02	字节	磁头号
0x03	字	开始减小写电流的柱面(仅 PC XT 使用，其他为 0)
0x05	字	开始写前预补偿柱面号 (乘 4)
0x07	字节	最大 ECC 猝发长度 (仅 XT 使用，其他为 0)
0x08	字节	控制字节 (驱动器步进选择)
		位 0 未用
		位 1 保留(0) (关闭 IRQ)

		位 2 允许复位 位 3 若磁头数大于 8 则置 1 位 4 未用(0) 位 5 若在柱面数+1 处有生产商的坏区图，则置 1 位 6 禁止 ECC 重试 位 7 禁止访问重试。
0x09	字节	标准超时值（仅 XT 使用，其他为 0）
0x0A	字节	格式化超时值（仅 XT 使用，其他为 0）
0x0B	字节	检测驱动器超时值（仅 XT 使用，其他为 0）
0x0C	字	磁头着陆(停止)柱面号
0x0E	字节	每磁道扇区数
0x0F	字节	保留。

9.4.3.4 硬盘设备号命名方式

硬盘的主设备号是 3。其他设备的主设备号分别为：
 1-内存,2-磁盘,3-硬盘,4-ttyx,5-tty,6-并行口,7-非命名管道
 由于 1 个硬盘中可以存在 1--4 个分区，因此硬盘还依据分区不同用次设备号进行指定分区。因此硬盘的逻辑设备号由以下方式构成：
 设备号=主设备号*256 + 次设备号
 也即 dev_no = (major<<8) + minor
 两个硬盘的所有逻辑设备号见表 9-11 所示。

表 9 - 11 硬盘逻辑设备号

逻辑设备号	对应设备文件	说明
0x300	/dev/hd0	代表整个第 1 个硬盘
0x301	/dev/hd1	表示第 1 个硬盘的第 1 个分区
0x302	/dev/hd2	表示第 1 个硬盘的第 2 个分区
0x303	/dev/hd3	表示第 1 个硬盘的第 3 个分区
0x304	/dev/hd4	表示第 1 个硬盘的第 4 个分区
0x305	/dev/hd5	代表整个第 2 个硬盘
0x306	/dev/hd6	表示第 2 个硬盘的第 1 个分区
0x307	/dev/hd7	表示第 2 个硬盘的第 2 个分区
0x308	/dev/hd8	表示第 2 个硬盘的第 3 个分区
0x309	/dev/hd9	表示第 2 个硬盘的第 4 个分区

其中 0x300 和 0x305 并不与哪个分区对应，而是代表整个硬盘。
 从 linux 内核 0.95 版后已经不使用这种烦琐的命名方式，而是使用与现在相同的命名方法了。

9.4.3.5 硬盘分区表

如果 PC 机从硬盘上引导启动操作系统，那么 ROM BIOS 程序在执行完机器自检诊断程序以后就会把硬盘上的第 1 个扇区读入内存 0x7c00 开始处，并把执行控制权交给这个扇区中的代码去继续执行。这个特定的扇区被称为主引导扇区 MBR（Master Boot Record），其结构见表 9-12 所示。

表 9-12 硬盘主引导扇区 MBR 的结构

偏移位置	名称	长度 (字节)	说明
0x000	MBR 代码	446	引导程序代码和数据。
0x1BE	分区表项 1	16	第 1 个分区表项, 共 16 字节。
0x1CE	分区表项 2	16	第 2 个分区表项, 共 16 字节。
0x1DE	分区表项 3	16	第 3 个分区表项, 共 16 字节。
0x1EE	分区表项 4	16	第 4 个分区表项, 共 16 字节。
0x1FE	引导标志	2	有效引导扇区的标志, 值分别是 0x55, 0xAA。

除了 446 字节的引导执行代码以外, MBR 中还包含一张硬盘分区表, 共含有 4 个表项。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE--0x1FD 偏移位置处。为了实现多个操作系统共享硬盘资源, 硬盘可以在逻辑上把所有扇区分成 1--4 个分区。每个分区之间的扇区号是邻接的。分区表中每个表项有 16 字节, 用来描述一个分区的特性。其中存放有分区的大小和起止的柱面号、磁道号和扇区号, 见表 9-13 所示。

表 9 - 13 硬盘分区表项结构

位置	名称	大小	说明
0x00	boot_ind	字节	引导标志。4 个分区中同时只能有一个分区是可引导的。 0x00-不从该分区引导操作系统; 0x80-从该分区引导操作系统。
0x01	head	字节	分区起始磁头号。磁头号范围为 0--255。
0x02	sector	字节	分区起始当前柱面中扇区号(位 0-5) (1--63) 和柱面号高 2 位(位 6-7)。
0x03	cyl	字节	分区起始柱面号低 8 位。柱面号范围为 0--1023。
0x04	sys_ind	字节	分区类型字节。0x0b-DOS; 0x80-Old Minix; 0x83-Linux ...
0x05	end_head	字节	分区结束处磁头号。磁头号范围为 0--255。
0x06	end_sector	字节	分区结束当前柱面中扇区号(位 0-5) (1--63) 和柱面号高 2 位(位 6-7)。
0x07	end_cyl	字节	分区结束柱面号低 8 位。柱面号范围为 0--1023。
0x08-0x0b	start_sect	长字	分区起始物理扇区号。从整个硬盘顺序计起的扇区号, 从 0 计起。
0x0c-0x0f	nr_sects	长字	分区占用的扇区数。

表中字段 head、sector 和 cyl 分别代表分区开始处的磁头号、柱面中扇区号和柱面号。磁头号的取值范围是 0--255。sector 字节字段中低 6 比特位代表在当前柱面中计数的扇区号, 该扇区号计数范围是 1--63。sector 字段高 2 比特与 cyl 字段组成 10 个比特的柱面号, 取值范围是 0--1023。类似地, 表中 end_head、end_sector 和 end_cyl 字段分别表示分区结束处的磁头号、柱面中扇区号和柱面号。因此若我们用 H 表示磁头号、S 表示扇区号、C 表示柱面号, 那么分区起始 CHS 值可表示为:

H = head;

S = sector & 0x3f;

C = (sector & 0xc0) << 2 + cyl;

表中 start_sect 字段是 4 个字节的分区起始物理扇区号。它表示整个硬盘从 0 计起的顺序编制的扇区号。编码方法是从 CHS 为 0 柱面、0 磁头和 1 扇区 (0, 0, 1) 开始, 先对当前柱面中扇区进行从小到大编码, 然后对磁头从 0 到最大磁头号编码, 最后是对柱面进行计数。

如果一个硬盘的磁头总数是 MAX_HEAD, 每磁道扇区总数是 MAX_SECT, 那么某个 CHS 值对应的硬盘物理扇区号 phy_sector 就是:

phy_sector = (C * MAX_HEAD + H) * MAX_SECT + S - 1

硬盘的第 1 个扇区（0 柱面 0 头 1 扇区）除了多包含一个分区表以外，在其他方面与软盘上第一个扇区（boot 扇区）的作用一样，只是它的代码会在执行时把自己从 0x7c00 下移到 0x6000 处，腾出 0x7c00 处的空间，然后根据分区表中的信息，找出活动分区是哪一个，接着把活动分区的第 1 个扇区加载到 0x7c00 处去执行。一个分区从硬盘的哪个柱面、磁头和扇区开始，都记录在分区表中。因此从分区表中可以知道一个活动分区的第 1 个扇区（即该分区的引导扇区）在硬盘的什么地方。

9.4.3.6 扇区号与柱面号、当前磁道扇区号和当前磁头号的对应关系

假定硬盘的每磁道扇区数是 `track_secs`，硬盘磁头总数是 `dev_heads`，指定的硬盘顺序扇区号是 `sector`，对应当前磁道总数是 `tracks`，对应的柱面号是 `cyl`，在当前磁道上的扇区号是 `sec`，磁头号是 `head`。那么若想从指定顺序扇区号 `sector` 换算成对应的柱面号、当前磁道上扇区号以及当前磁头号，则可以使用以下步骤：

- `sector / track_secs` = 整数是 `tracks`，余数是 `sec`；
- `tracks / dev_heads` = 整数是 `cyl`，余数是 `head`；
- 在当前磁道上扇区号从 1 算起，于是需要把 `sec` 增 1。

若想从指定的当前 `cyl`、`sec` 和 `head` 换算成从硬盘开始算起的顺序扇区号，则过程正好与上述相反。换算公式和上面给出的完全一样，即：

$$\text{sector} = (\text{cyl} * \text{dev_heads} + \text{head}) * \text{track_secs} + \text{sec} - 1$$

9.5 ll_rw_blk.c 程序

9.5.1 功能描述

该程序主要用于执行低层块设备读/写操作，是本章所有块设备与系统其他部分的接口程序。其他程序通过调用该程序的低级块读写函数 `ll_rw_block()` 来读写块设备中的数据。该函数的主要功能是为块设备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请求项处理函数 `request_fn()` 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()`。若 `ll_rw_block()` 为一个块设备建立起一个请求项，并通过测试块设备的当前请求项指针为空而确定设备空闲时，就会设置该新建的请求项为当前请求项，并直接调用 `request_fn()` 对该请求项进行操作。否则就会使用电梯算法将新建的请求项插入到该设备的请求项链表中等待处理。而当 `request_fn()` 结束对一个请求项的处理，就会把该请求项从链表中删除。

由于 `request_fn()` 在每个请求项处理结束时，都会通过中断回调 C 函数（主要是 `read_intr()` 和 `write_intr()`）再次调用 `request_fn()` 自身去处理链表中其余的请求项，因此，只要设备的请求项链表（或者称为队列）中有未处理的请求项存在，都会陆续地被处理，直到设备的请求项链表是空为止。当请求项链表空时，`request_fn()` 将不再向驱动器控制器发送命令，而是立刻退出。因此，对 `request_fn()` 函数的循环调用就此结束。参见图 9-5 所示。

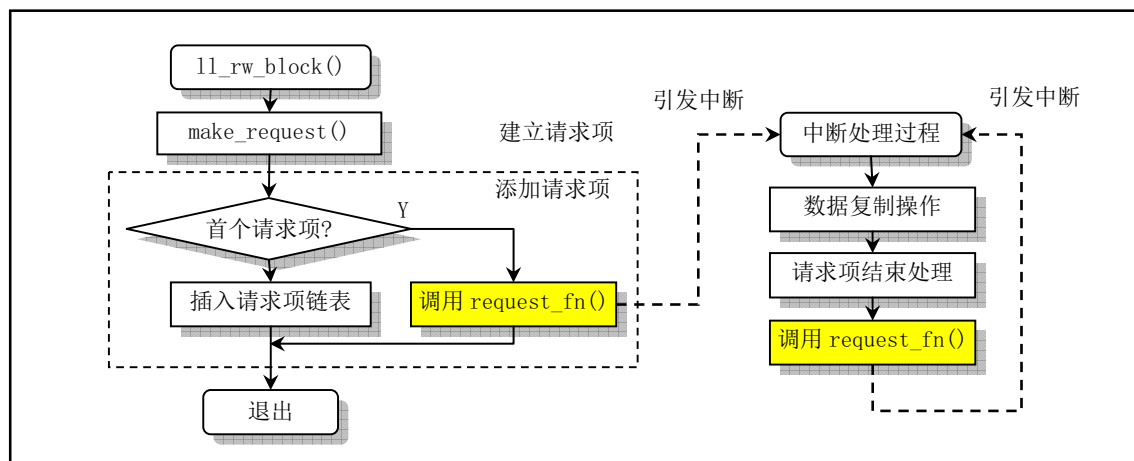


图 9-5 ll_rw_block 调用序列

对于虚拟盘设备，由于它的读写操作不牵涉到上述与外界硬件设备同步操作，因此没有上述的中断处理过程。当前请求项对虚拟设备的读写操作完全在 do_rd_request()中实现。

9.5.2 代码注释

程序 9-4 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 /*
11  * 该程序处理块设备的所有读/写操作。
12 */
13 #include <errno.h> // 错误号头文件。包含系统中各种出错号。
14 #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。
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/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。
17
18 #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。
19
20 /*
21  * The request-struct contains all necessary data
22  * to load a nr of sectors into memory
23 */
24 /*
25  * 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。
26 */
27 // 请求项数组队列。共有 NR_REQUEST = 32 个请求项。
28 struct request request[NR_REQUEST];
29

```

```

23 /*
24  * used to wait on when there are no free requests
25  */
26 /*
27  * 是用于在请求数组没有空闲项时进程的临时等待处。
28  */
29 struct task_struct * wait_for_request = NULL;
30
31 /* blk_dev_struct is:
32  *      do_request-address
33  *      next-request
34  */
35 /* blk_dev_struct 块设备结构是：（参见文件 kernel/blk_drv/blk.h，第 45 行）
36  *      do_request-address      // 对应主设备号的请求处理程序指针。
37  *      current-request        // 该设备的下一个请求。
38  */
39 // 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。
40 // 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。
41 struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
42     { NULL, NULL },          /* no_dev */    // 0 - 无设备。
43     { NULL, NULL },          /* dev mem */   // 1 - 内存。
44     { NULL, NULL },          /* dev fd */    // 2 - 软驱设备。
45     { NULL, NULL },          /* dev hd */    // 3 - 硬盘设备。
46     { NULL, NULL },          /* dev ttyx */  // 4 - ttyx 设备。
47     { NULL, NULL },          /* dev tty */   // 5 - tty 设备。
48     { NULL, NULL },          /* dev lp */    // 6 - lp 打印机设备。
49 };
50
51 // 锁定指定缓冲块。
52 // 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
53 // 锁缓冲块的任务明确地唤醒。
54 static inline void lock_buffer(struct buffer_head * bh)
55 {
56     cli();                      // 清中断许可。
57     while (bh->b_lock)           // 如果缓冲区已被锁定则睡眠，直到缓冲区解锁。
58         sleep_on(&bh->b_wait);
59     bh->b_lock=1;                // 立刻锁定该缓冲区。
60     sti();                      // 开中断。
61 }
62
63 // 释放（解锁）锁定的缓冲区。
64 // 该函数与 blk.h 文件中的同名函数完全一样。
65 static inline void unlock_buffer(struct buffer_head * bh)
66 {
67     if (!bh->b_lock)             // 如果该缓冲区没有被锁定，则打印出错信息。
68         printk("ll_rw_block.c: buffer not locked\n");
69     bh->b_lock = 0;              // 清锁定标志。
70     wake_up(&bh->b_wait);        // 唤醒等待该缓冲区的任务。
71 }
72
73 /*
74  * add-request adds a request to the linked list.
75  * It disables interrupts so that it can muck with the

```

```

62  * request-lists in peace.
63  */
/*
 * 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 */
*/
///// 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。
64 static void add_request(struct blk_dev_struct * dev, struct request * req)
65 {
66     struct request * tmp;
67
// 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。
68     req->next = NULL;
69     cli(); // 关中断。
70     if (req->bh)
71         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。
72     if (!(tmp = dev->current_request)) {
73         dev->current_request = req;
74         sti(); // 开中断。
75         (dev->request_fn)(); // 执行请求函数，对于硬盘是 do_hd_request()。
76         return;
77     }
// 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
// 当前请求插入到请求链表中。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。
78     for (; tmp->next; tmp=tmp->next)
79         if ((IN_ORDER(tmp, req) ||
80             !IN_ORDER(tmp, tmp->next)) &&
81             IN_ORDER(req, tmp->next))
82             break;
83     req->next=tmp->next;
84     tmp->next=req;
85     sti();
86 }
87
///// 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。
88 static void make_request(int major, int rw, struct buffer_head * bh)
89 {
90     struct request * req;
91     int rw_ahead; // 逻辑值，用于判断是否为 READA 或 WRITEA 命令。

```

```

92
93 /* WRITEA/READA is special case - it is not really needed, so if the */
94 /* 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。
95     if (rw_ahead = (rw == READA || rw == WRITEA)) {
96         if (bh->b_lock)
97             return;
98         if (rw == READA)
99             rw = READ;
100         else
101             rw = WRITE;
102     }
103     if (rw!=READ && rw!=WRITE)
104         panic("Bad block dev command, must be R/W/RA/WA");
    // 对命令 rw 进行了一番处理之后, 现在只有 READ 或 WRITE 两种命令。在开始生成和添加相
    // 应读/写数据请求项之前, 我们再来看看此次是否有必要添加请求项。在两种情况下可以不
    // 必添加请求项。一是当命令是写 (WRITE), 但缓冲区中的数据在读入之后并没有被修改过;
    // 二是当命令是读 (READ), 但缓冲区中的数据已经是更新过的, 即与块设备上的完全一样。
    // 因此这里首先锁定缓冲区对其检查一下。如果此时缓冲区已被上锁, 则当前任务就会睡眠,
    // 直到被明确地唤醒。如果确实是属于上述两种情况, 那么就可以直接解锁缓冲区, 并返回。
    // 这几行代码体现了高速缓冲区的用意, 在数据可靠的情况下就无须再执行硬盘操作, 而直接
    // 使用内存中的现有数据。缓冲块的 b_dirt 标志用于表示缓冲块中的数据是否已经被修改过。
    // b_uptodate 标志用于表示缓冲块中的数据是与块设备上的同步, 即在从块设备上读入缓冲块
    // 后没有修改过。
105     lock\_buffer(bh);
106     if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
107         unlock\_buffer(bh);
108         return;
109     }
110 repeat:
111 /* we don't allow the write-requests to fill up the queue completely:
112 * we want some room for reads: they take precedence. The last third
113 * of the requests are only for reads.
114 */
    /* 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。
115     if (rw == READ)
116         req = request+NR\_REQUEST;           // 对于读请求, 将指针指向队列尾部。
117     else

```

```

118         req = request+(NR_REQUEST*2)/3); // 对于写请求, 指针指向队列 2/3 处。
119 /* find an empty request */                /* 搜索一个空请求项 */
120         while (--req >= request)
121             if (req->dev<0)
122                 break;
123 /* if none found, sleep on new requests: check for rw_ahead */
124 /* 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 */
125         if (req < request) {                // 如果已搜索到头(队列无空项),
126             if (rw_ahead) {                // 则若是提前读/写请求就退出。
127                 unlock_buffer(bh);
128                 return;
129             }
130             sleep_on(&wait_for_request);    // 否则就睡眠, 过会再查看请求队列。
131             goto repeat;                    // 跳转 110 行去重新搜索。
132 }
133 /* fill up the request-info, and add it to the queue */
134 /* 向空闲请求项中填写请求信息, 并将其加入队列中 */
135 // 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
136 // 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。
137 // 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。
138 req->dev = bh->b_dev;                        // 设备号。
139 req->cmd = rw;                              // 命令(READ/WRITE)。
140 req->errors=0;                              // 操作时产生的错误次数。
141 req->sector = bh->b_blocknr<<1;              // 起始扇区。块号转换成扇区号(1 块=2 扇区)。
142 req->nr_sectors = 2;                        // 本请求项需要读写的扇区数。
143 req->buffer = bh->b_data;                    // 请求项缓冲区指针指向需读写的数据缓冲区。
144 req->waiting = NULL;                        // 任务等待操作执行完成的地方。
145 req->bh = bh;                              // 缓冲块头指针。
146 req->next = NULL;                          // 指向下一请求项。
147 add_request(major+blk_dev, req);           // 将请求项加入队列中(blk_dev[major], req)。
148 }
149
150 // 低层读写数据块函数 (Low Level Read Write Block)。
151 // 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。
152 // 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
153 // 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
154 // 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
155 // 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。
156 // 参数: rw - READ、READA、WRITE 或 WRITEA 是命令; bh - 数据缓冲块头指针。
157 void ll_rw_block(int rw, struct buffer_head * bh)
158 {
159     unsigned int major;                    // 主设备号 (对于硬盘是 3)。
160
161     // 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。
162     // 否则创建请求项并插入请求队列。
163     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
164         !(blk_dev[major].request_fn)) {
165         printk("Trying to read nonexistent block-device\n");
166         return;
167     }
168     make_request(major, rw, bh);
169 }

```

```

//// 块设备初始化函数，由初始化程序 main.c 调用（init/main.c, 128）。
// 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)。
157 void blk_dev_init(void)
158 {
159     int i;
160
161     for (i=0 ; i<NR_REQUEST ; i++) {
162         request[i].dev = -1;
163         request[i].next = NULL;
164     }
165 }
166

```

9.6 ramdisk.c 程序

9.6.1 功能描述

本文件是内存虚拟盘（Ram Disk）驱动程序，由 Theodore Ts'o 编制。虚拟盘设备是一种利用物理内存来模拟实际磁盘存储数据的方式。其目的主要是为了提高对“磁盘”数据的读写操作速度。除了需要占用一些宝贵的内存资源外，其主要缺点是一旦系统崩溃或关闭，虚拟盘中的所有数据将全部消失。因此虚拟盘中通常存放一些系统命令等常用工具程序或临时数据，而非重要的输入文档。

当在 linux/Makefile 文件中定义了常量 RAMDISK，内核初始化程序就会在内存中划出一块该常量值指定大小的内存区域用于存放虚拟盘数据。虚拟盘在物理内存中所处的具体位置是在内核初始化阶段确定的（init/main.c, 123 行），它位于内核高速缓冲区和主内存区之间。若运行的机器含有 16MB 的物理内存，那么虚拟盘区域会被设置在内存 4MB 开始处，虚拟盘容量即等于 RAMDISK 的值（KB）。若 RAMDISK=512，则此时内存情况见图 9-6 所示。

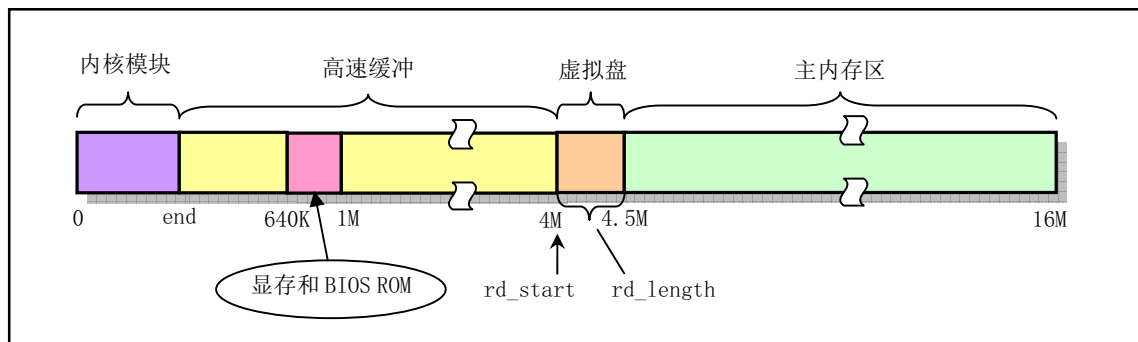


图 9-6 虚拟盘在 16MB 内存系统中所处的具体位置

对虚拟盘设备的读写访问操作原则上完全按照对普通磁盘的操作进行，也需要按照块设备的访问方式对其进行读写操作。由于在实现上不牵涉与外部控制器或设备进行同步操作，因此其实现方式比较简单。对于数据在系统与设备之间的“传送”只需执行内存数据块复制操作即可。

本程序包含 3 个函数。rd_init() 会在系统初始化时被 init/main.c 程序调用，用于确定虚拟盘在物理内存中的具体位置和大小；do_rd_request() 是虚拟盘设备的请求项操作函数，对当前请求项实现虚拟盘数据的访问操作；rd_load() 是虚拟盘根文件加载函数。在系统初始化阶段，该函数被用于尝试从启动引导盘

上指定的磁盘块位置开始处把一个根文件系统加载到虚拟盘中。在函数中，这个起始磁盘块位置被定为 256。当然你也可以根据自己的具体要求修改这个值，只要保证这个值所规定的磁盘容量能容纳内核映像文件即可。这样一个由内核引导映像文件（Bootimage）加上根文件系统映像文件（Rootimage）组合而成的“二合一”磁盘，就可以象启动 DOS 系统盘那样来启动 Linux 系统。关于这种组合盘（集成盘）的制作方式可参见第 14 章中相关内容。

在进行正常的根文件系统加载之前，系统会首先执行 `rd_load()` 函数，试图从磁盘的第 257 块中读取根文件系统超级块。若成功，就把该根文件映像文件读到内存虚拟盘中，并把根文件系统设备标志 `ROOT_DEV` 设置为虚拟盘设备（0x0101），否则退出 `rd_load()`，系统继续从别的设备上执行根文件加载操作。操作流程见图 9-7 所示。

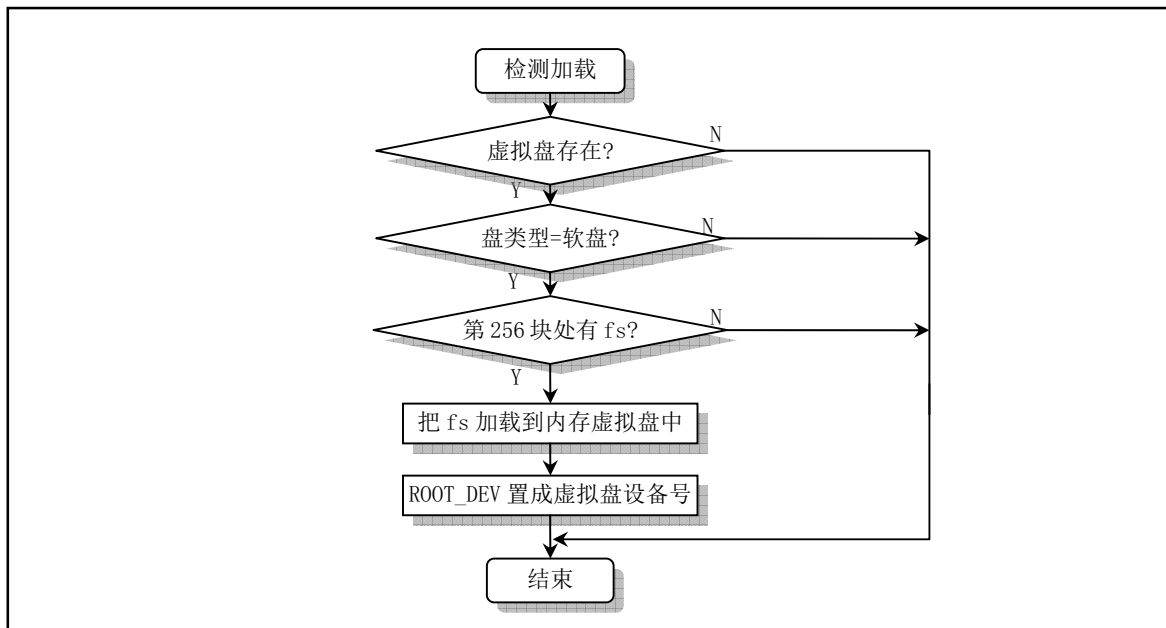


图 9-7 加载根文件系统到内存虚拟盘区域的流程图

如果在编译 Linux 0.11 内核源代码时，在其 `linux/Makefile` 配置文件中定义了 `RAMDISK` 的大小值，则内核代码在引导并初始化 `RAMDISK` 区域后就会首先尝试检测启动盘上的第 256 磁盘块（每个磁盘块为 1KB，即 2 个扇区）开始处是否存在一个根文件系统。检测方法是判断第 257 磁盘块中是否存在一个有效的文件系统超级块信息。如果有，则将该文件系统加载到 `RAMDISK` 区域中，并将其作为根文件系统使用。从而我们就可以使用一张集成了根文件系统的启动盘来引导系统到 `shell` 命令提示符状态。若启动盘上指定磁盘块位置（第 256 磁盘块）上没有存放一个有效的根文件系统，那么内核就会提示插入根文件系统盘。在用户按下回车键确认后，内核就把处于独立盘上的根文件系统整个地读入到内存的虚拟盘区域中去执行。

在一张 1.44MB 的内核引导启动盘上把一个基本的根文件系统放在盘的第 256 个磁盘块开始的地方就可以组合形成一张集成盘，其结构见图 9-8 所示。

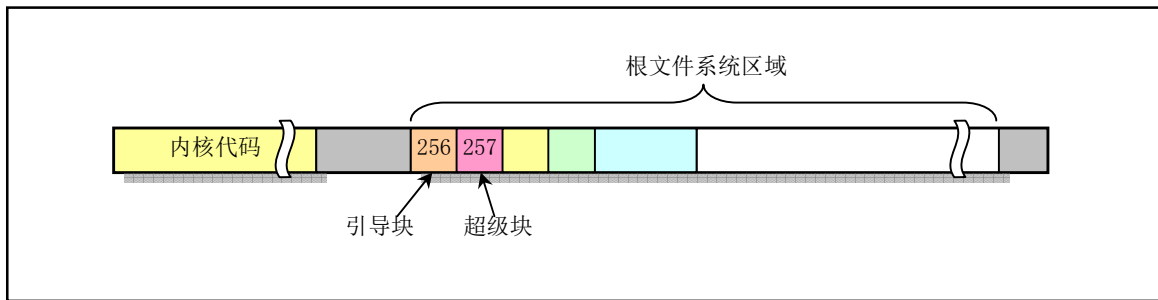


图 9-8 集成盘上数据结构

9.6.2 代码注释

程序 9-5 linux/kernel/blk_drv/ramdisk.c

```

1  /*
2   * linux/kernel/blk_drv/ramdisk.c
3   *
4   * Written by Theodore Ts'o, 12/2/91
5   */
   /* 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-ll.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/)
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
9  #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。
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/fs.h>        // 文件系统头文件。定义文件表结构 (file、m_inode) 等。
12 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原型定义。
13 #include <asm/system.h>      // 系统头文件。定义了设置或修改描述符/中断门等嵌入式汇编宏。
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 #include <asm/memory.h>      // 内存拷贝头文件。含有 memcpy() 嵌入式汇编宏函数。
16
   // 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。
17 #define MAJOR_NR 1
18 #include "blk.h"
19
   // 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。
20 char    *rd_start;           // 虚拟盘在内存中的开始地址。
21 int      rd_length = 0;       // 虚拟盘所占内存大小（字节）。

```

```

22 // 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。
23 void do_rd_request(void)
24 {
25     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，L124 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
80     // 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
81     // 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。
82     if (!rd_length)
83         return;
84     printk("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
85         (int) rd_start);
86     if (MAJOR(ROOT_DEV) != 2)
87         return;
88
89     // 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
90     // 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
91     // 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
92     // 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
93     // 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
94     // 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。
95     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 */
95         /* 磁盘中没有 ramdisk 映像文件, 退出去执行通常的软盘引导 */
96         return;
97     // 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
98     // 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
99     // 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
100    // 数 * 2^(每区段块数的次方)), 即 nblocks = (s_nzones * 2^s_log_zone_size)。如果遇到
101    // 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
102    // 能显示出错信息并返回。
103    nblocks = s.s_nzones << s.s_log_zone_size;
104    if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
105        printk("Ram disk image too big! (%d blocks, %d avail)\n",
106            nblocks, rd_length >> BLOCK_SIZE_BITS);
107        return;
108    }
109    // 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
110    // 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。
111    // 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(),
112    // 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
113    // 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
114    // 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。
115    printk("Loading %d bytes into ram disk... 0000k",
116        nblocks << BLOCK_SIZE_BITS);
117    cp = rd_start;
118    while (nblocks) {
119        if (nblocks > 2) // 若读取块数多于 2 块则采用超前预读。
120            bh = breada(ROOT_DEV, block, block+1, block+2, -1);
121        else // 否则就单块读取。
122            bh = bread(ROOT_DEV, block);
123        if (!bh) {
124            printk("I/O error on block %d, aborting load\n",
125                block);
126            return;
127        }
128        (void) memcpy(cp, bh->b_data, BLOCK_SIZE); // 复制到 cp 处。
129        brelse(bh);
130        printk(" \010\010\010\010\010%4dk", i); // 打印加载块计数值。
131        cp += BLOCK_SIZE; // 虚拟盘指针前移。
132        block++;
133        nblocks--;
134        i++;
135    }
136    // 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
137    // 根文件设备号修改成虚拟盘的设备号 0x0101, 最后返回。
138    printk(" \010\010\010\010\010done \n");
139    ROOT_DEV=0x0101;

```

125 }
126

9.7 floppy.c 程序

9.7.1 功能描述

本程序是软盘控制器驱动程序。与其他块设备驱动程序一样，该程序也以请求项操作函数 `do_fd_request()` 为主，执行对软盘上数据的读写操作。

考虑到软盘驱动器在不工作时马达通常不转，所以在实际能对驱动器中的软盘进行读写操作之前，我们需要等待马达启动并达到正常的运行速度。与计算机的运行速度相比，这段时间较长，通常需要 0.5 秒左右的时间。

另外，当对一个磁盘的读写操作完毕，我们也需要让驱动器停止转动，以减少对磁盘表面的摩擦。但我们也不能在对磁盘操作完后就立刻让它停止转动。因为，可能马上又需要对其进行读写操作。因此，在一个驱动器没有操作后还是需要通过空转一段时间，以等待可能到来的读写操作，若驱动器在一个较长时间内都没有操作，则程序让它停止转动。这段维持旋转的时间可设定在大约 3 秒左右。

当一个磁盘的读写操作发生错误，或某些其他情况导致一个驱动器的马达没有被关闭。此时我们也需要让系统在一段时间之后自动将其关闭。Linux 在程序中把这个延时值设定在 100 秒。

由此可见，在对软盘驱动器进行操作时会用到很多延时（定时）操作。因此在该驱动程序中涉及较多的定时处理函数。还有几个与定时处理关系比较密切的函数被放在了 `kernel/sched.c` 中（行 201-262）。这是软盘驱动程序与硬盘驱动程序之间的最大区别，也是软盘驱动程序比硬盘驱动程序复杂的原因。

虽然本程序比较复杂，但对软盘读写操作的工作原理却与其他块设备是一样的。本程序也是使用请求项和请求项链表结构来处理所有对软盘的读写操作。因此请求项操作函数 `do_fd_request()` 仍然是本程序中的重要函数之一。在阅读时应该以该函数为主线展开。另外，软盘控制器的使用比较复杂，其中涉及到很多控制器的执行状态和标志。因此在阅读时，还需要频繁地参考程序后的有关说明以及本程序的头文件 `include/linux/fdreg.h`。该文件定义了所有软盘控制器参数常量，并说明了这些常量的含义。

9.7.2 代码注释

程序 9-6 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 */
13
14 /*
15  * 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
16  * 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
17  * 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  */
21 /*
22  * 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，
23  * 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。
24  * 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，
25  * 但不能保证问题已消失。
26  */
27 // 英文注释中的“Urgel”是瑞典语，对应英文单词“Ugh”。该单词在最新内核中还常出现。
28 // 2003年9月份Linux曾说过出现这个词的地方就表示相关的代码有问题或比较差劲，例如
29 // 这个软盘驱动程序 :)
30
31 /*
32  * As with hd.c, all routines within this file can (and will) be called
33  * by interrupts, so extreme caution is needed. A hardware interrupt
34  * handler may not sleep, or a kernel panic will happen. Thus I cannot
35  * call "floppy-on" directly, but have to set a special timer interrupt
36  * etc.
37  *
38  * Also, I'm not certain this works on more than 1 floppy. Bugs may
39  * abund.
40  */
41 /*
42  * 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
43  * 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
44  * 直接调用“floppy-on”，而只能设置一个特殊的定时中断等。
45  *
46  * 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。
47  */
48
49 #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，
50 // 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。
51 #include <linux/fs.h> // 文件系统头文件。含文件表结构 (file, buffer_head, m_inode) 等。
52 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
53 #include <linux/fdreg.h> // 软驱头文件。含有软盘控制器参数的一些定义。
54 #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。
55 #include <asm/io.h> // io头文件。定义硬件端口输入/输出宏汇编语句。
56 #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。
57
58 // 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。
59 // 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。
60 #define MAJOR_NR 2 // 软驱的主设备号是2。
61 #include "blk.h" // 块设备头文件。定义请求数据结构、块设备结构和宏函数等信息。
62
63 static int recalibrate = 0; // 标志：1表示需要重新校正磁头位置（磁头归零道）。
64 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, 204 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  * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
97  * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
98  * H is head unload time (1=16ms, 2=32ms, etc)
99  *
100 * Spec2 is (HLD<<1 | ND), where HLD is head load time (1=2ms, 2=4 ms etc)
101 * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
102 */
/*
* 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。
103 // floppy_interrupt() 是 system_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 457 行) 使用它初始化中断陷阱门描述符。
104 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。
105 extern char tmp_floppy_area[1024];
106
107 /*
108  * These are global variables, as that's the easiest way to give

```

```

109 * information to interrupts. They are the data used for the current
110 * request.
111 */
/*
* 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。
112 static int cur_spec1 = -1; // 当前软盘参数 spec1。
113 static int cur_rate = -1; // 当前软盘转速 rate。
114 static struct floppy_struct * floppy = floppy_type; // 软盘类型结构数组指针。
115 static unsigned char current_drive = 0; // 当前驱动器号。
116 static unsigned char sector = 0; // 当前扇区号。
117 static unsigned char head = 0; // 当前磁头号。
118 static unsigned char track = 0; // 当前磁道号。
119 static unsigned char seek_track = 0; // 寻道磁道号。
120 static unsigned char current_track = 255; // 当前磁头所在磁道号。
121 static unsigned char command = 0; // 命令。
122 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。
123 struct task_struct * wait_on_floppy_select = NULL; // 等待选定软驱的任务队列。
124
///// 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)。
125 void floppy_deselect(unsigned int nr)
126 {
127     if (nr != (current_DOR & 3))
128         printk("floppy_deselect: drive not selected\n\r");
129     selected = 0; // 复位软驱已选定标志。
130     wake_up(&wait_on_floppy_select); // 唤醒等待的任务。
131 }
132
133 /*
134 * floppy-change is never called from an interrupt, so we can relax a bit
135 * here, sleep etc. Note that floppy-on tries to set current_DOR to point
136 * to the desired drive, but it will probably not survive the sleep if
137 * several floppies are used at the same time: thus the loop.
138 */
/*
* 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)。
139 int floppy_change(unsigned int nr)
140 {
// 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。
141 repeat:
142     floppy_on(nr);          // 启动并等待指定软驱 nr (kernel/sched.c, 第 232 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。
143     while ((current_DOR & 3) != nr && selected)
144         interruptible_sleep_on(&wait_on_floppy_select);
145     if ((current_DOR & 3) != nr)
146         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。
147     if (inb(FD_DIR) & 0x80) {
148         floppy_off(nr);
149         return 1;
150     }
151     floppy_off(nr);
152     return 0;
153 }
154
///// 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。
155 #define copy_buffer(from,to) \
156 __asm__ ("cld ; rep ; movsl" \
157         ::"c" (BLOCK_SIZE/4), "S" ((long)(from)), "D" ((long)(to)) \
158         : "cx", "di", "si")
159
///// 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。
160 static void setup_DMA(void)
161 {
162     long addr = (long) CURRENT->buffer;          // 当前请求项缓冲区所处内存地址。
163
// 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。
164     cli();
165     if (addr >= 0x100000) {
166         addr = (long) tmp_floppy_area;
167         if (command == FD_WRITE)
168             copy_buffer(CURRENT->buffer, tmp_floppy_area);
169     }
// 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。

```

```

170 /* mask DMA 2 */ /* 屏蔽 DMA 通道 2 */
171     immoutb_p(4|2, 10);
172 /* output command byte. I don't know why, but everyone (minix, */
173 /* sanches & canton) output this twice, first to 12 then to 11 */
174     /* 输出命令字节。我是不知道为什么，但是每个人 (minix, */
175     /* sanches 和 canton) 都输出两次，首先是 12 口，然后是 11 口 */
176     // 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
177     // 方式字（读盘时是 0x46，写盘是 0x4A）。
178     // 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。
179     // 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。
180     // 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，
181     // 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
182     // 器的设置从低字节开始。
183     asm ("outb %%al, $12\n\tjmp 1f\n1:\tjmp 1f\n1:\t"
184          "outb %%al, $11\n\tjmp 1f\n1:\tjmp 1f\n1:"
185          "a" ((char) ((command == FD_READ)?DMA_READ:DMA_WRITE)));
186 /* 8 low bits of addr */ /* 地址低 0-7 位 */
187     // 向 DMA 通道 2 写入基/当前地址寄存器（端口 4）。
188     immoutb_p(addr, 4);
189     addr >>= 8;
190 /* bits 8-15 of addr */ /* 地址高 8-15 位 */
191     immoutb_p(addr, 4);
192     addr >>= 8;
193 /* bits 16-19 of addr */ /* 地址 16-19 位 */
194     // 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）。
195     immoutb_p(addr, 0x81);
196 /* low 8 bits of count-1 (1024-1=0x3ff) */ /* 计数器低 8 位(1024-1 = 0x3ff) */
197     // 向 DMA 通道 2 写入基/当前字节计数值（端口 5）。
198     immoutb_p(0xff, 5);
199 /* high 8 bits of count-1 */ /* 计数器高 8 位 */
200     // 一次共传输 1024 字节（两个扇区）。
201     immoutb_p(3, 5);
202 /* activate DMA 2 */ /* 开启 DMA 通道 2 的请求 */
203     immoutb_p(0|2, 10);
204     sti();
205 }
206
207 //// 向软驱控制器输出一个字节命令或参数。
208 // 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
209 // 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
210 // 作适当延时。若出错，则会设置复位标志 reset。
211 static void output_byte(char byte)
212 {
213     int counter;
214     unsigned char status;
215
216     if (reset)
217         return;
218     // 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
219     // 方向位 STATUS_DIR = 0 (CPU→FDC)，则向数据端口输出指定字节。
220     for(counter = 0 ; counter < 10000 ; counter++) {
221         status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
222         if (status == STATUS_READY) {

```

```

204         outb(byte, FD_DATA);
205         return;
206     }
207 }
// 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。
208     reset = 1;
209     printk("Unable to send byte to FDC\n\r");
210 }
211
///// 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。
212 static int result(void)
213 {
214     int i = 0, counter, status;
215
// 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。
216     if (reset)
217         return -1;
218     for (counter = 0 ; counter < 10000 ; counter++) {
219         status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
// 如果控制器状态是 READY，表示已经没有数据可取，则返回已读取的字节数 i。
220         if (status == STATUS_READY)
221             return 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。
222         if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
223             if (i >= MAX_REPLIES)
224                 break;
225             reply_buffer[i++] = inb_p(FD_DATA);
226         }
227     }
// 如果到循环 1 万次结束时还不能发送，则置复位标志，并打印出错信息。
228     reset = 1;
229     printk("Getstatus times out\n\r");
230     return -1;
231 }
232
///// 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。
233 static void bad_flp_intr(void)
234 {
// 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）。
235     CURRENT->errors++;
236     if (CURRENT->errors > MAX_ERRORS) {
237         floppy_deselect(current_drive);
238         end_request(0);

```



```

239     }
    // 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。
240     if (CURRENT->errors > MAX_ERRORS/2)
241         reset = 1;
242     else
243         recalibrate = 1;
244 }
245
246 /*
247  * Ok, this interrupt is called after a DMA read/write has succeeded,
248  * so we check the results, and copy any buffers.
249  */
    /*
    * 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。
250 static void rw_interrupt(void)
251 {
    // 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 )
252     if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
253         if (ST1 & 0x02) { // 0x02 = ST1_WP - Write Protected.
254             printk("Drive %d is write protected\n\r",current drive);
255             floppy_deselect(current drive);
256             end_request(0);
257         } else
258             bad_flp_intr();
259         do_fd_request();
260         return;
261     }
    // 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）。
262     if (command == FD_READ && (unsigned long)(CURRENT->buffer) >= 0x100000)
263         copy_buffer(tmp_floppy_area,CURRENT->buffer);
    // 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。
264     floppy_deselect(current drive);
265     end_request(1);
266     do_fd_request();
267 }
268
    // 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。
269 inline void setup_rw_floppy(void)
270 {
271     setup_DMA();           // 初始化软盘 DMA 通道。
272     do_floppy = rw_interrupt; // 置软盘中断调用函数指针。
273     output_byte(command);   // 发送命令字节。
274     output_byte(head<<2 | current_drive); // 参数: 磁头号+驱动器号。
275     output_byte(track);     // 参数: 磁道号。
276     output_byte(head);      // 参数: 磁头号。
277     output_byte(sector);    // 参数: 起始扇区号。
278     output_byte(2);         /* sector size = 512 */ // 参数: (N=2) 512 字节。
279     output_byte(floppy->sect); // 参数: 每磁道扇区数。
280     output_byte(floppy->gap);  // 参数: 扇区间隔长度。
281     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。
282     if (reset)
283         do_fd_request();
284 }
285
286 /*
287  * This is the routine called after every seek (or recalibrate) interrupt
288  * from the floppy controller. Note that the "unexpected interrupt" routine
289  * also does a recalibrate, but doesn't come here.
290  */
291 /*
292  * 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
293  * "unexpected interrupt" (意外中断) 子程序也会执行重新校正操作, 但不在此地。
294  */
295 // 寻道处理结束后中断过程中调用的 C 函数。
296 // 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
297 // 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
298 // 用函数 setup_rw_floppy() 设置 DMA 并输出软盘读写命令和参数。
299 static void seek_interrupt(void)
300 {
301     // 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
302     // 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
303     // 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
304     // 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。
305     /* sense drive status */ /* 检测驱动器状态 */
306     output_byte(FD_SENSEI);
307     if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
308         bad_flp_intr();
309         do_fd_request();
310         return;
311     }
312     // 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。
313     current_track = ST1; // 设置当前磁道。
314     setup_rw_floppy();   // 设置 DMA 并输出软盘操作命令和参数。
315 }
316

```

```

304 /*
305  * This routine is called when everything should be correctly set up
306  * for the transfer (ie floppy motor is on and the correct floppy is
307  * selected).
308  */
/*
 * 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。
309 static void transfer(void)
310 {
// 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)。
311     if (cur_spec1 != floppy->spec1) {           // 检测当前参数。
312         cur_spec1 = floppy->spec1;
313         output_byte(FD_SPECIFY);           // 发送设置磁盘参数命令。
314         output_byte(cur_spec1);           /* hut etc */ // 发送参数。
315         output_byte(6);                     /* Head load time =6ms, DMA */
316     }
317     if (cur_rate != floppy->rate)           // 检测当前速率。
318         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。
319     if (reset) {
320         do_fd_request();
321         return;
322     }
// 如果此时若寻道标志为零（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。
323     if (!seek) {
324         setup_rw_floppy();           // 发送命令参数块。
325         return;
326     }
327     do_floppy = seek_interrupt;       // 寻道中断调用函数。
328     if (seek_track) {                 // 起始磁道号。
329         output_byte(FD_SEEK);           // 发送磁头寻道命令。
330         output_byte(head<<2 | current_drive); // 发送参数：磁头号+当前软驱号。
331         output_byte(seek_track);       // 发送参数：磁道号。
332     } else {
333         output_byte(FD_RECALIBRATE);     // 发送重新校正命令（磁头归零）。
334         output_byte(head<<2 | current_drive); // 发送参数：磁头号+当前软驱号。
335     }
// 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。
336     if (reset)
337         do_fd_request();

```

```

338 }
339
340 /*
341  * Special case - used after a unexpected interrupt (or reset)
342  */
343 /*
344  * 特殊情况 - 用于意外中断（或复位）处理后。
345  */
346 // 软驱重新校正中断调用函数。
347 // 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
348 // 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。
349 static void recal_interrupt(void)
350 {
351     output_byte(FD SENSEI); // 发送检测中断状态命令。
352     if (result() != 2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
353         reset = 1; // 异常结束，则置复位标志。
354     else // 否则复位重新校正标志。
355         recalibrate = 0;
356     do_fd_request(); // 作相应处理。
357 }
358
359 // 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。
360 // 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
361 // 校正标志。
362 void unexpected_floppy_interrupt(void)
363 {
364     output_byte(FD SENSEI); // 发送检测中断状态命令。
365     if (result() != 2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
366         reset = 1; // 异常结束，则置复位标志。
367     else // 否则置重新校正标志。
368         recalibrate = 1;
369 }
370
371 // 软盘重新校正处理函数。
372 // 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
373 // 重新校正命令就会再其引发的软盘中断中调用 recal_interrupt() 函数。
374 static void recalibrate_floppy(void)
375 {
376     recalibrate = 0; // 复位重新校正标志。
377     current_track = 0; // 当前磁道号归零。
378     do_floppy = recal_interrupt; // 指向重新校正中断调用的 C 函数。
379     output_byte(FD RECALIBRATE); // 命令：重新校正。
380     output_byte(head << 2 | current_drive); // 参数：磁头号 + 当前驱动器号。
381     // 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
382     // 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()
383     // 中的复位处理代码。
384     if (reset)
385         do_fd_request();
386 }
387
388 // 软盘控制器 FDC 复位中断调用函数。
389 // 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。
390 // 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。
373 static void reset_interrupt(void)
374 {
375     output_byte(FD_SENSEI);           // 发送检测中断状态命令。
376     (void) result();                 // 读取命令执行结果字节。
377     output_byte(FD_SPECIFY);         // 发送设定软驱参数命令。
378     output_byte(cur_spec1);          /* hut etc */ // 发送参数。
379     output_byte(6);                  /* Head load time =6ms, DMA */
380     do_fd_request();                 // 调用执行软盘请求。
381 }
382
383 /*
384  * reset is done by pulling bit 2 of DOR low for a while.
385  */
/* 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。
386 static void reset_floppy(void)
387 {
388     int i;
389
390     reset = 0;                       // 复位标志置 0。
391     cur_spec1 = -1;                  // 使无效。
392     cur_rate = -1;
393     recalibrate = 1;                 // 重新校正标志置位。
394     printf("Reset-floppy called\n\r"); // 显示执行软盘复位操作信息。
395     cli();                           // 关中断。
396     do_floppy = reset_interrupt;     // 设置在中断处理程序中调用的函数。
397     outb_p(current_DOR & ~0x04, FD_DOR); // 对软盘控制器 FDC 执行复位操作。
398     for (i=0; i<100; i++)             // 空操作，延迟。
399         __asm__("nop");
400     outb(current_DOR, FD_DOR);        // 再启动软盘控制器。
401     sti();                            // 开中断。
402 }
403
///// 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()。
404 static void floppy_on_interrupt(void) // floppy_on() interrupt.
405 {
406     /* 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。
407     selected = 1; // 置已选定当前驱动器标志。
408     if (current_drive != (current_DOR & 3)) {
409         current_DOR &= 0xFC;
410         current_DOR |= current_drive;
411         outb(current_DOR, FD_DOR); // 向数字输出寄存器输出当前 DOR。
412         add_timer(2, &transfer); // 添加定时器并执行传输函数。
413     } else
414         transfer(); // 执行软盘读写传输函数。
415 }
416
///// 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。
417 void do_fd_request(void)
418 {
419     unsigned int block;
420
// 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。
421     seek = 0; // 清寻道标志。
422     if (reset) { // 复位标志已置位。
423         reset_floppy();
424         return;
425     }
426     if (recalibrate) { // 重新校正标志已置位。
427         recalibrate_floppy();
428         return;
429     }
// 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。
430     INIT_REQUEST;
431     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。
432     if (current_drive != CURRENT_DEV) // CURRENT_DEV 是请求项中指定的软驱号。
433         seek = 1;
434     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。
435     block = CURRENT->sector; // 取当前软盘请求项中起始扇区号。
436     if (block+2 > floppy->size) { // 如果 block + 2 大于磁盘扇区总数，
437         end_request(0); // 则结束本次软盘请求项。
438         goto repeat;
439     }
// 再求对应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）。

```

```

440     sector = block % floppy->sect;    // 起始扇区对每磁道扇区数取模，得磁道上扇区号。
441     block /= floppy->sect;             // 起始扇区对每磁道扇区数取整，得起始磁道数。
442     head = block % floppy->head;       // 起始磁道数对磁头数取模，得操作的磁头号。
443     track = block / floppy->head;      // 起始磁道数对磁头数取整，得操作的磁道号。
444     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。
445     if (seek_track != current_track)
446         seek = 1;
447     sector++;                         // 磁盘上实际扇区计数是从 1 算起。
448     if (CURRENT->cmd == READ)         // 如果请求项是读操作，则置读命令码。
449         command = FD_READ;
450     else if (CURRENT->cmd == WRITE)    // 如果请求项是写操作，则置写命令码。
451         command = FD_WRITE;
452     else
453         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()。
454     add_timer(ticks_to_floppy_on(current_drive), &floppy_on_interrupt);
455 }
456
///// 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。
457 void floppy_init(void)
458 {
// 设置软盘中断门描述符。floppy_interrupt (kernel/system_call.s, 252 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。
459     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST; // = do_fd_request()。
460     set_trap_gate(0x26, &floppy_interrupt);       // 设置陷阱门描述符。
461     outb(inb_p(0x21) & ~0x40, 0x21);              // 复位软盘中断请求屏蔽位。
462 }
463

```

9.7.3 其他信息

9.7.3.1 软盘驱动器的设备号

在 Linux 中，软驱的主设备号是 2，次设备号 = TYPE*4 + DRIVE，其中 DRIVE 为 0-3，分别对应软驱 A、B、C 或 D；TYPE 是软驱的类型，2 表示 1.2M 软驱，7 表示 1.44M 软驱，也即 floppy.c 中 85 行定义的软盘类型 (floppy_type[]) 数组的索引值，见表 9-14 所示。

表 9-14 软盘驱动器类型

类型	说明
0	不用。
1	360KB PC 软驱。
2	1.2MB AT 软驱。
3	360kB 在 720kB 驱动器中使用。

4	3.5" 720kB 软盘。
5	360kB 在 1.2MB 驱动器中使用。
6	720kB 在 1.2MB 驱动器中使用。
7	1.44MB 软驱。

例如, 类型 7 表示 1.44MB 驱动器, 驱动器号 0 表示 A 盘, 因为 $7 \times 4 + 0 = 28$, 所以(2,28)指的是 1.44M A 驱动器, 其设备号是 0x021c, 对应的设备文件名是/dev/fd0 或/dev/PS0。同理, 类型 2 表示 1.22MB 驱动器, 则 $2 \times 4 + 0 = 8$, 所以(2,8)指的是 1.2M A 驱动器, 其设备号是 0x0208, 对应的设备文件名是/dev/at0。

9.7.3.2 软盘控制器

对软盘控制器 (FDC) 进行编程比较烦琐。在编程时需要访问 4 个端口, 分别对应软盘控制器上一个或多个寄存器。对于 1.2M 的软盘控制器有表 9-15 中的一些端口。

表 9-15 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器 (DOR) (数字控制寄存器)
0x3f4	只读	FDC 主状态寄存器 (STATUS)
0x3f5	读/写	FDC 数据寄存器 (DATA)
0x3f7	只读	数字输入寄存器 (DIR)
	只写	磁盘控制寄存器 (DCR) (传输率控制)

数字输出端口 DOR (数字控制端口) 是一个 8 位寄存器, 它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。该寄存器各比特位的含义见表 9-16 所示。

表 9-16 数字输出寄存器定义

位	名称	说明
7	MOT_EN3	启动软驱 D 马达: 1-启动; 0-关闭。
6	MOT_EN2	启动软驱 C 马达: 1-启动; 0-关闭。
5	MOT_EN1	启动软驱 B 马达: 1-启动; 0-关闭。
4	MOT_EN0	启动软驱 A 马达: 1-启动; 0-关闭。
3	DMA_INT	允许 DMA 和中断请求; 0-禁止 DMA 和中断请求。
2	RESET	允许软盘控制器 FDC 工作。0-复位 FDC。
1	DRV_SEL1	00-11 用于选择软盘驱动器 A-D。
0	DRV_SEL0	

FDC 的主状态寄存器也是一个 8 位寄存器, 用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常, 在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前, 都要读取主状态寄存器的状态位, 以判别当前 FDC 数据寄存器是否就绪, 以及确定数据传送的方向。见表 9-17 所示。

表 9-17 FDC 主状态控制器 MSR 定义

位	名称	说明
7	RQM	数据口就绪: 控制器 FDC 数据寄存器已准备就绪。
6	DIO	传输方向: 1- FDC→CPU; 0- CPU→FDC

5	NDM	非 DMA 方式: 1- 非 DMA 方式; 0- DMA 方式
4	CB	控制器忙: FDC 正处于命令执行忙碌状态
3	DDB	软驱 D 忙
2	DCB	软驱 C 忙
1	DBB	软驱 B 忙
0	DAB	软驱 A 忙

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

数据输入寄存器（DIR）只有位 7（D7）对软盘有效，用来表示盘片更换状态。其余七位用于硬盘控制器接口。

磁盘控制寄存器(DCR)用于选择盘片在不同类型驱动器上使用的数据传输率。仅使用低 2 位(D1D0)，00 表示 500kbps，01 表示 300kbps，10 表示 250kbps。

Linux 0.11 内核中，驱动程序与软驱中磁盘之间的数据传输是通过 DMA 控制器实现的。在进行读写操作之前，需要首先初始化 DMA 控制器，并对软驱控制器进行编程。对于 386 兼容 PC，软驱控制器使用硬件中断 IR6（对应中断描述符 0x26），并采用 DMA 控制器的通道 2。有关 DMA 控制处理的内容见后面小节。

9.7.3.3 软盘控制器命令

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0--8 字节的参数。

执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。

结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0--7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

由于 Linux 0.11 的软盘驱动程序中只使用其中 6 条命令，因此这里仅对这些用到的命令进行描述。

1. 重新校正命令（FD_RECALIBRATE）

该命令用来让磁头退回到 0 磁道。通常用于在软盘操作出错时对磁头重新校正定位。其命令码是 0x07，参数是指定的驱动器号（0—3）。

该命令无结果阶段，程序需要通过执行“检测中断状态”来获取该命令的执行结果。见表 9-18 所示。

表 9-18 重新校正命令（FD_RECALIBRATE）

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	0	1	1	1	重新校正命令码: 0x07

	1	0	0	0	0	0	0	US1	US2	驱动器号
执行										磁头移动到 0 磁道
结果										需使用命令获取执行结果。

2. 磁头寻道命令 (FD_SEEK)

该命令让选中驱动器的磁头移动到指定磁道上。第 1 个参数指定驱动器号和磁头号，位 0-1 是驱动器号，位 2 是磁头号，其他比特位无用。第 2 个参数指定磁道号。

该命令也无结果阶段，程序需要通过执行“检测中断状态”来获取该命令的执行结果。见表 9-19 所示。

表 9 - 19 磁头寻道命令 (FD_SEEK)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	1	1	1	1	磁头寻道命令码：0x0F
	1	0	0	0	0	0	HD	US1	US2	磁头号、驱动器号。
	2	C								磁道号。
执行										磁头移动到指定磁道上。
结果										需使用命令获取执行结果。

3. 读扇区数据命令 (FD_READ)

该命令用于从磁盘上读取指定位置开始的扇区，经 DMA 控制传输到系统内存中。每当一个扇区读完，参数 4 (R) 就自动加 1，以继续读取下一个扇区，直到 DMA 控制器把传输计数终止信号发送给软盘控制器。该命令通常是在磁头寻道命令执行后磁头已经位于指定磁道后开始。见表 9-20 所示。

返回结果中，磁道号 C 和扇区号 R 是当前磁头所处位置。因为在读完一个扇区后起始扇区号 R 自动增 1，因此结果中的 R 值是下一个未读扇区号。若正好读完一个磁道上最后一个扇区（即 EOT），则磁道号也会增 1，并且 R 值复位成 1。

表 9 - 20 读扇区数据命令 (FD_READ)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	MT	MF	SK	0	0	1	1	0	读命令码：0xE6 (MT=MF=SK=1)
	1	0	0	0	0	0	0	US1	US2	驱动器号。
	2	C								磁道号
	3	H								磁头号
	4	R								起始扇区号
	5	N								扇区字节数
	6	EOT								磁道上最大扇区号
	7	GPL								扇区之间间隔长度 (3)
	8	DTL								N=0 时，指定扇区字节数
执行										数据从磁盘传送到系统
结果	1	ST0								状态字节 0
	2	ST1								状态字节 1
	3	ST2								状态字节 2
	4	C								磁道号
	5	H								磁头号

6	R	扇区号
7	N	扇区字节数

其中 MT、MF 和 SK 的含义分别为：

MT 表示多磁道操作。MT=1 表示允许在同一磁道上两个磁头连续操作。

MF 表示记录方式。MF=1 表示选用 MFM 记录方式，否则是 FM 记录方式。

SK 表示是否跳过有删除标志的扇区。SK=1 表示跳过。

返回的 3 个状态字节 ST0、ST1 和 ST2 的含义分别见表 9-21、表 9-22 和表 9-23 所示。

表 9-21 状态字节 0 (ST0)

位	名称	说明
7	ST0_INTR	中断原因。00 – 命令正常结束；01 – 命令异常结束； 10 – 命令无效；11 – 轮流查询操作而导致的异常终止。
6		
5	ST0_SE	寻道操作或重新校正操作结束。(Seek End)
4	ST0_ECE	设备检查出错（零磁道校正出错）。(Equip. Check Error)
3	ST0_NR	软驱未就绪。(Not Ready)
2	ST0_HA	磁头地址。中断时磁头号。(Head Address)
1	ST0_DS	驱动器选择号（发生中断时驱动器号）。(Drive Select) 00 – 11 分别对应驱动器 0—3。
0		

表 9-22 状态字节 1 (ST1)

位	名称	说明
7	ST1_EOC	访问超过磁道上最大扇区号 EOT。(End of Cylinder)
6		未使用 (0)。
5	ST1_CRC	CRC 校验出错。
4	ST1_OR	数据传输超时，DMA 控制器故障。(Over Run)
3		未使用 (0)。
2	ST1_ND	未找到指定的扇区。(No Data - unreadable)
1	ST1_WP	写保护。(Write Protect)
0	ST1_MAM	未找到扇区地址标志 ID AM。(Missing Address Mask)

表 9-23 状态字节 2 (ST2)

位	名称	说明
7		未使用 (0)。
6	ST2_CM	SK=0 时，读数据遇到删除标志。(Control Mark = deleted)
5	ST2_CRC	扇区数据场 CRC 校验出错。
4	ST2_WC	扇区 ID 信息的磁道号 C 不符。(Wrong Cylinder)
3	ST2_SEH	检索（扫描）条件满足要求。(Scan Equal Hit)
2	ST2_SNS	检索条件不满足要求。(Scan Not Satisfied)
1	ST2_BC	扇区 ID 信息的磁道号 C=0xFF，磁道坏。(Bad Cylinder)
0	ST2_MAM	未找到扇区数据标志 DATA AM。(Missing Address Mask)

4. 写扇区数据命令 (FD_WRITE)

该命令用于将内存中的数据写到磁盘上。在 DMA 传输方式下，软驱控制器把内存中的数据串行地写到磁盘指定扇区中。每写完一个扇区，起始扇区号自动增 1，并继续写下一个扇区，直到软驱控制器收到 DMA 控制器的计数终止信号。见表 9-24 所示，其中缩写名称的含义与读命令中的相同。

表 9-24 写扇区数据命令 (FD_WRITE)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	MT	MF	0	0	0	1	0	1	写数据命令码：0xC5 (MT=MF=1)
	1	0	0	0	0	0	0	US1	US2	驱动器号。
	2	C								磁道号
	3	H								磁头号
	4	R								起始扇区号
	5	N								扇区字节数
	6	EOT								磁道上最大扇区号
	7	GPL								扇区之间间隔长度 (3)
	8	DTL								N=0 时，指定扇区字节数
执行										数据从系统传送到磁盘
结果	1	ST0								状态字节 0
	2	ST1								状态字节 1
	3	ST2								状态字节 2
	4	C								磁道号
	5	H								磁头号
	6	R								扇区号
	7	N								扇区字节数

5. 检测中断状态命令 (FD_SENSEI)

发送该命令后软驱控制器会立刻返回常规结果 1 和 2 (即状态 ST0 和磁头所处磁道号 PCN)。它们是控制器执行上一条命令后的状态。通常在一个命令执行结束后会向 CPU 发出中断信号。对于读写扇区、读写磁道、读写删除标志、读标识场、格式化和扫描等命令以及非 DMA 传输方式下的命令引起的中断，可以直接根据主状态寄存器的标志知道中断原因。而对于驱动器就绪信号发生变化、寻道和重新校正 (磁头回零道) 而引起的中断，由于没有返回结果，就需要利用本命令来读取控制器执行命令后的状态信息。见表 9-25 所示。

表 9-25 检测中断状态命令 (FD_SENSEI)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	1	0	0	0	检测中断状态命令码：0x08
执行										
结果	1	ST0								状态字节 0
	2	C								磁头所在磁道号

6. 设定驱动器参数命令 (FD_SPECIFY)

该命令用于设定软盘控制器内部的三个定时器初始值和选择传输方式，即把驱动器马达步进速率 (SRT)、磁头加载/卸载 (HLT/HUT) 时间和是否采用 DMA 方式来传输数据的信息送入软驱控制器。见表 9-26 所示。其中时间单位是当数据传输率为 500KB/S 时的值。另外，在 Linux 0.11 内核中，命令

阶段的序 1 字节即是 floppy.c 文件中第 95 行下英文注释中说明的 spec1 参数；序 2 字节是 spec2 参数。由该英文注释和参考第 315 行上的程序语句可知，spec2 被固定设置成值 6（即 HLT=3，ND=0），表示磁头加载时间是 6 毫秒，使用 DMA 方式。

表 9-26 设定驱动器参数命令 (FD_SPECIFY)

阶段	序	D7	D6	D5	D4	D3	D2	D1	D0	说明
命令	0	0	0	0	0	0	0	1	1	设定参数命令码：0x03
	1	SRT（单位 1ms）				HUT（单位 16ms）				马达步进速率、磁头卸载时间
	2	HLT（单位 2ms）							ND	磁头加载时间、非 DMA 方式
执行										设置控制器，不发生中断
结果		无								无

9.7.3.4 软盘控制器编程方法

在 PC 机中，软盘控制器一般采用与 NEC PD765 或 Intel 8287A 兼容的芯片，例如 Intel 的 82078。由于软盘的驱动程序比较复杂，因此下面对这类芯片构成的软盘控制器的编程方法进行较为详细的介绍。

典型的磁盘操作不仅仅包括发送命令和等待控制器返回结果，的软盘驱动器的控制是一种低级操作，它需要程序在不同阶段对其执行状况进行干涉。

◆ 命令与结果阶段的交互

在上述磁盘操作命令或参数发送到软盘控制器之前，必须首先查询控制器的主状态寄存器（MSR），以获知驱动器的就绪状态和数据传输方向。软盘驱动程序中使用了一个 output_byte(byte)函数来专门实现该操作。该函数的等效框图见图 9-9 所示。

该函数一直循环到主状态寄存器的数据口就绪标志 RQM 为 1，并且方向标志 DIO 是 0（CPU→FDC），此时控制器就已准备好接受命令和参数字节。循环语句起超时计数功能，以应付控制器没有响应的情况。本驱动程序中把循环次数设置成了 10000 次。对这个循环次数的选择需要仔细，以避免程序作出不正确的超时判断。在 Linux 内核版本 0.1x 至 0.9x 中就经常会碰到需要调整这个循环次数的问题，因为当时人们所使用的 PC 机运行速度差别较大（16MHz -- 40MHz），因此循环所产生的实际延时也有很大的区别。这可以参见早期 Linux 的邮件列表中的许多文章。为了彻底解决这个问题，最好能使用系统硬件时钟来产生固定频率的延时值。

对于读取控制器的结果字节串的结果阶段，也需要采取与发送命令相同的操作方法，只是此时数据传输方向标志要求是置位状态（FDC→CPU）。本程序中对应的函数是 result()。该函数把读取的结果状态字节存放到了 reply_buffer[] 字节数组中。

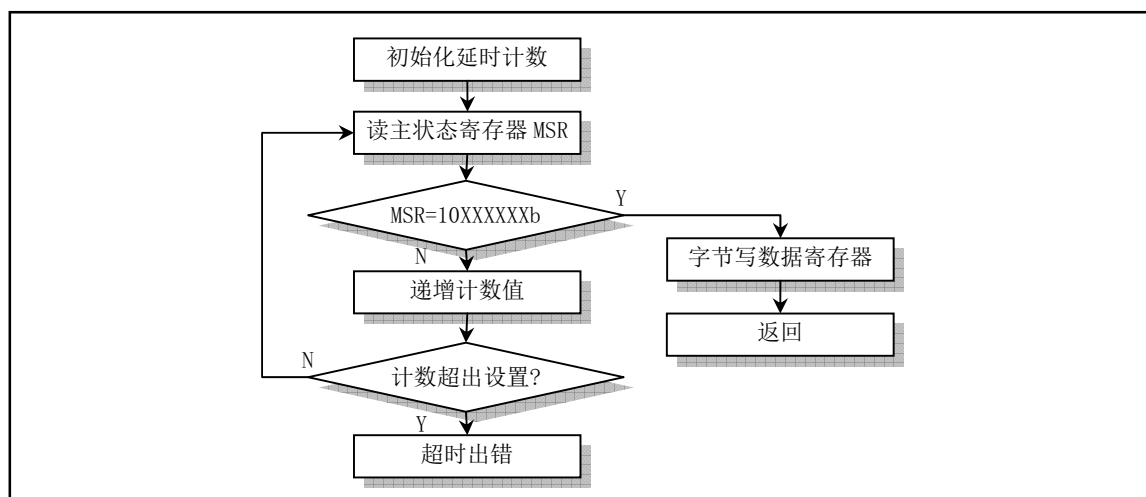


图 9-9 向软盘控制器发送命令或参数字节

◆软盘控制器初始化

对软盘控制器的初始化操作包括在控制器复位后对驱动器进行适当的参数配置。控制器复位操作是指对数字输出寄存器 DOR 的位 2（启动 FDC 标志）置 0 然后再置 1。在机器复位之后，“指定驱动器参数”命令 SPECIFY 所设置的值就不再有效，需要重新建立。在 floppy.c 程序中，复位操作在函数 reset_floppy() 和中断处理 C 函数 reset_interrupt() 中。前一个函数用于修改 DOR 寄存器的位 2，让控制器复位，后一个函数用于在控制器复位后使用 SPECIFY 命令重新建立控制器中的驱动器参数。在数据传输准备阶段，若判断出与实际的磁盘规格不同，还在传输函数 transfer() 开始处对其另行进行重新设置。

在控制器复位后，还应该向数字控制寄存器 DCR 发送指定的传输速率值，以重新初始化数据传输速率。如果机器执行了复位操作（例如热启动），则数据传输速率会变成默认值 250Kpbs。但通过数字输出寄存器 DOR 向控制器发出的复位操作并不会影响设置的数据传输速率。

◆驱动器重新校正和磁头寻道

驱动器重新校正（FD_RECALIBRATE）和磁头寻道（FD_SEEK）是两个磁头定位命令。重新校正命令让磁头移动到零磁道，而磁头寻道命令则让磁头移动到指定的磁道上。这两个磁头定位命令与典型的读/写命令不同，因为它们没有结果阶段。一旦发出这两个命令之一，控制器将立刻会在主状态寄存器（MSR）返回就绪状态，并以后台形式执行磁头定位操作。当定位操作完成后，控制器就会产生中断以请求服务。此时就应该发送一个“检测中断状态”命令，以结束中断和读取定位操作后的状态。由于驱动器和马达启动信号是直接由数字输出寄存器（DOR）控制的，因此，如果驱动器或马达还没有启动，那么写 DOR 的操作必须在发出定位命令之前进行。流程图见图 9-10 所示。

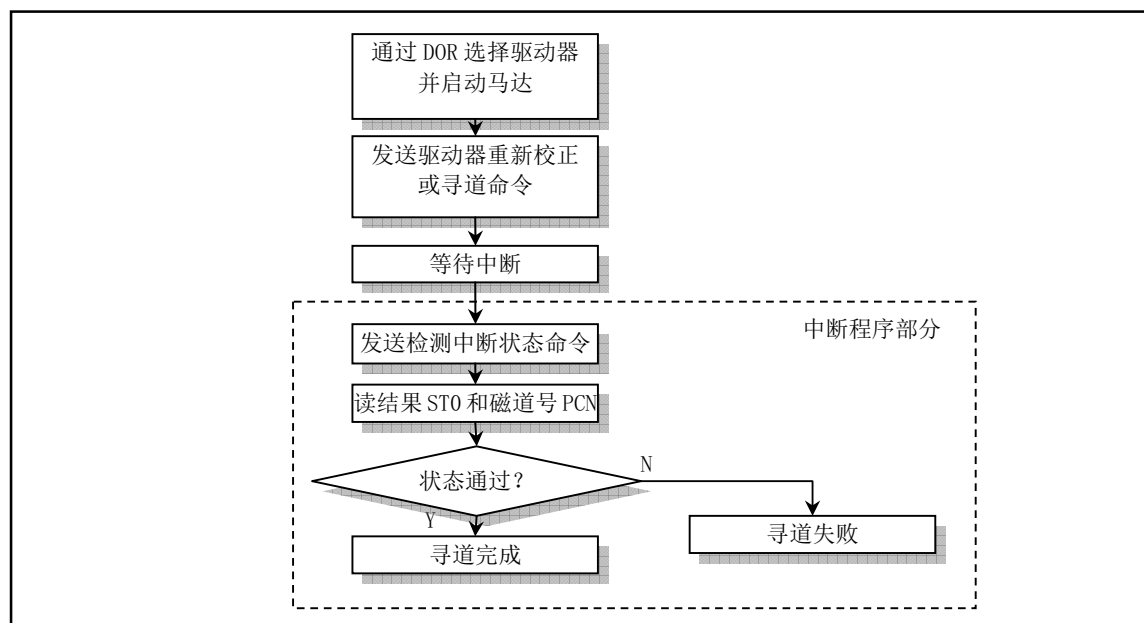


图 9-10 重新校正和寻道操作

◆数据读/写操作

数据读或写操作需要分几步来完成。首先驱动器马达需要开启，并把磁头定位到正确的磁道上，然后初始化 DMA 控制器，最后发送数据读或写命令。另外，还需要定出发生错误时的处理方案。典型的操作流程图见图 9-11 所示。

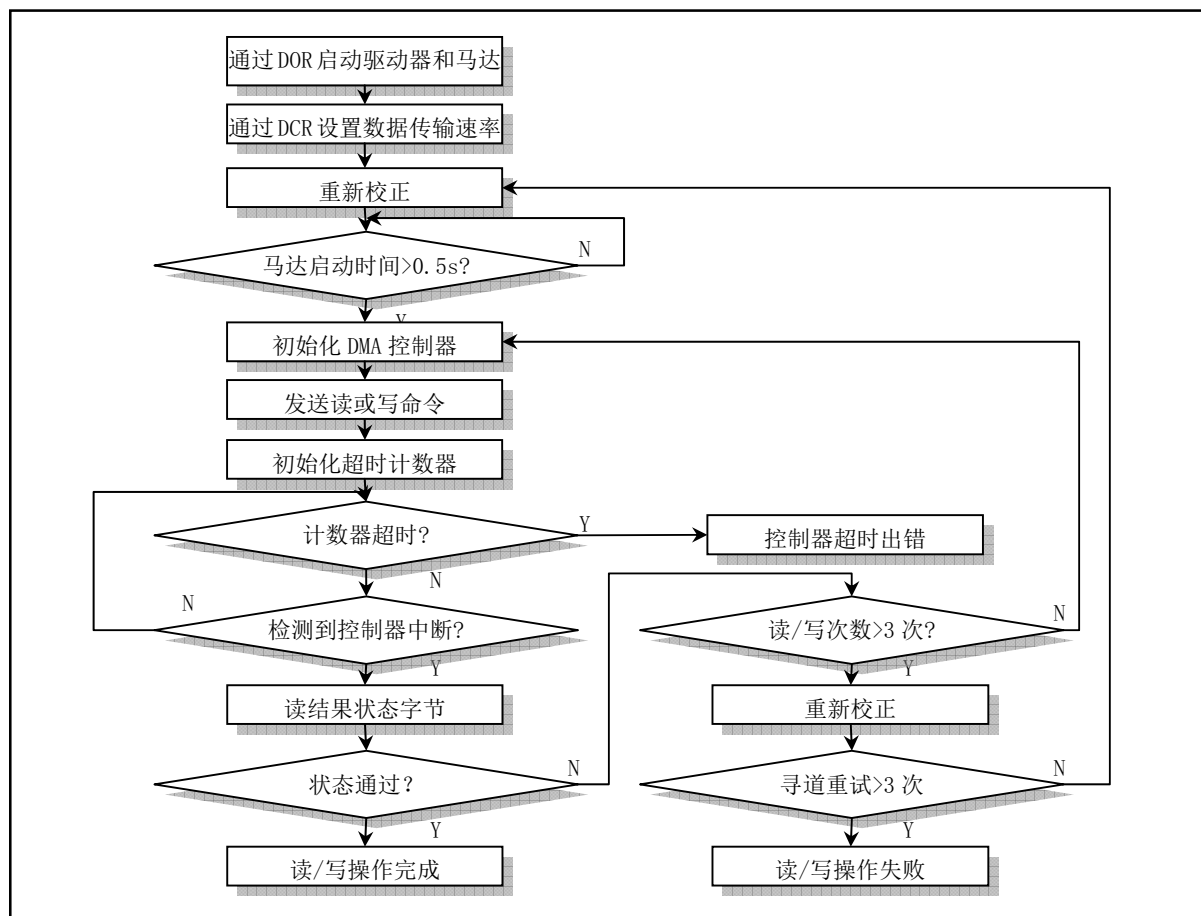


图 9-11 数据读/写操作流程

在对磁盘进行数据传输之前，磁盘驱动器的马达必须首先达到正常的运转速度。对于大多数 3 $\frac{1}{2}$ 英寸软驱来讲，这段启动时间大约需要 300ms，而 5 $\frac{1}{4}$ 英寸的软驱则需要大约 500ms。在 floppy.c 程序中将这个启动延迟时间设置成了 500ms。

在马达启动后，就需要使用数字控制寄存器 DCR 设置与当前磁盘介质匹配的数据传输率。

如果隐式寻道方式没有开启，接下来就需要发送寻道命令 FD_SEEK，把磁头定位到正确的磁道上。在寻道操作结束后，磁头还需要花费一段到位（加载）时间。对于大多数驱动器，这段延迟时间起码需要 15ms。当使用了隐式寻道方式，那么就可以使用“指定驱动器参数”命令指定的磁头加载时间（HLT）来确定最小磁头到位时间。例如在数据传输速率为 500Kbps 的情况下，若 HLT=8，则有效磁头到位时间是 16ms。当然，如果磁头已经在正确的磁道上到位了，也就无须确保这个到位时间了。

然后对 DMA 控制器进行初始化操作，读写命令也随即执行。通常，在数据传输完成后，DMA 控制器会发出终止计数（TC）信号，此时软盘控制器就会完成当前数据传输并发出中断请求信号，表明操作已到达结果阶段。如果在操作过程中出现错误或者最后一个扇区号等于磁道最后一个扇区（EOT），那么软盘控制器也会马上进入结果阶段。

根据上面流程图，如果在读取结果状态字节后发现错误，则会通过重新初始化 DMA 控制器，再尝试重新开始执行数据读或写操作命令。持续的错误通常表明寻道操作并没有让磁头到达指定的磁道，此时应该多次重复对磁头执行重新校准，并再次执行寻道操作。若此后还是出错，则最终控制器就会向驱动程序报告读写操作失败。

◆ 磁盘格式化操作

Linux 0.11 内核中虽然没有实现对软盘的格式化操作，但作为参考，这里还是对磁盘格式化操作进行简单说明。磁盘格式化操作过程包括把磁头定位到每个磁道上，并创建一个用于组成数据字段（场¹¹）的固定格式字段。

在马达已启动并且设置了正确的数据传输率之后，磁头会返回零磁道。此时磁盘需要在 500ms 延迟时间内到达正常和稳定的运转速度。

在格式化操作期间磁盘上建立的标识字段（ID 字段）是在执行阶段由 DMA 控制器提供。DMA 控制器被初始化成为每个扇区标识场提供磁道（C）、磁头（H）、扇区号（R）和扇区字节数的值。例如，对于每个磁道具有 9 个扇区的磁盘，每个扇区大小是 2（512 字节），若是用磁头 1 格式化磁道 7，那么 DMA 控制器应该被编程为传输 36 个字节的数据（9 扇区 x 每扇区 4 个字节），数据字段应该是：7,1,1,2, 7,1,2,2, 7,1,3,2, ..., 7,1,9,2。因为在格式化命令执行期间，软盘控制器提供的数据会被直接作为标识字段记录在磁盘上，数据的内容可以是任意的。因此有些人就利用这个功能来防止保护磁盘复制。

在一个磁道上的每个磁头都已经执行了格式化操作以后，就需要执行寻道操作让磁头前移到下一磁道上，并重复执行格式化操作。因为“格式化磁道”命令不含有隐式的寻道操作，所以必须使用寻道命令 SEEK。同样，前面所讨论的磁头到位时间也需要在每次寻道后设置。

9.7.3.5 DMA 控制器编程

DMA (Direct Memory Access)是“直接存储器访问”的缩写。DMA 控制器的主要功能是通过让外部设备直接与内存传输数据来增强系统的性能。通常它由机器上的 Intel 8237 芯片或其兼容芯片实现。通过对 DMA 控制器进行编程，外设与内存之间的数据传输能在不受 CPU 控制的条件下进行。因此在数据传输期间，CPU 可以做其他事。DMA 控制器传输数据的工作过程如下：

1. 初始化 DMA 控制器。

程序通过 DMA 控制器端口对其进行初始化操作。该操作包括：① 向 DMA 控制器发送控制命令；② 传输的内存起始地址；③ 数据长度。发送的命令指明传输使用的 DMA 通道、是内存传输到外设（写）还是外设数据传输到内存、是单字节传输还是批量（块）传输。对于 PC 机，软盘控制器被指定使用 DMA 通道 2。在 Linux 0.11 内核中，软盘驱动程序采用的是单字节传输模式。由于 Intel 8237 芯片只有 16 根地址引脚（其中 8 根与数据线合用），因此只能寻址 64KB 的内存空间。为了能让它访问 1MB 的地址空间，DMA 控制器采用了一个页面寄存器把 1MB 内存分成了 16 个页面来操作，见表 9-27 所示。因此传输的内存起始地址需要转换成所处的 DMA 页面值和页面中的偏移地址。每次传输的数据长度也不能超过 64KB。

表 9 - 27 DMA 页面对应的内存地址范围

DMA 页面	地址范围（64KB）
0x0	0x00000 - 0x0FFFF
0x1	0x10000 - 0x1FFFF
0x2	0x20000 - 0x2FFFF
0x3	0x30000 - 0x3FFFF
0x4	0x40000 - 0x4FFFF
0x5	0x50000 - 0x5FFFF
0x6	0x60000 - 0x6FFFF
0x7	0x70000 - 0x7FFFF
0x8	0x80000 - 0x8FFFF
0x9	0x90000 - 0x9FFFF

¹¹ 关于磁盘格式的说明资料，以前均把 filed 翻译成场。其实对于程序员来讲，翻译成字段或域或许更顺耳一些。☺

0xA	0xA0000 - 0xAFFFF
0xB	0xB0000 - 0xBFFFF
0xC	0xC0000 - 0xCFFFF
0xD	0xD0000 - 0xDFFFF
0xE	0xE0000 - 0xEFFFF
0xF	0xF0000 - 0xFFFFF

2. 数据传输

在初始化完成之后，对 DMA 控制器的屏蔽寄存器进行设置，开启 DMA 通道 2，从而 DMA 控制器开始进行数据的传输。

3. 传输结束

当所需传输的数据全部传输完成，DMA 控制器就会产生“操作完成”（EOP）信号发送到软盘控制器。此时软盘控制器即可执行结束操作：关闭驱动器马达并向 CPU 发送中断请求信号。

在 PC/AT 机中，DMA 控制器有 8 个独立的通道可使用，其中后 4 个通道是 16 位的。软盘控制器被指定使用 DMA 通道 2。在使用一个通道之前必须首先对其设置。这牵涉到对三个端口的操作，分别是：页面寄存器端口、（偏移）地址寄存器端口和数据计数寄存器端口。由于 DMA 寄存器是 8 位的，而地址和计数值是 16 位的，因此各自需要发送两次。首先发送低字节，然后发送高字节。每个通道对应的端口地址见表 9-28 所示。

表 9 - 28 DMA 各通道使用的页面、地址和计数寄存器端口

DMA 通道	页面寄存器	地址寄存器	计数寄存器
0	0x87	0x00	0x01
1	0x83	0x02	0x03
2	0x81	0x04	0x05
3	0x82	0x06	0x07
4	0x8F	0xC0	0xC2
5	0x8B	0xC4	0xC6
6	0x89	0xC8	0xCA
7	0x8A	0xCC	0xCE

对于通常的 DMA 应用，有 4 个常用寄存器用于控制 DMA 控制器的状态。它们是命令寄存器、请求寄存器、单屏蔽寄存器、方式寄存器和清除字节指针触发器。见表 9-29 所示。Linux 0.11 内核使用了表中带阴影的 3 个寄存器端口（0x0A, 0x0B, 0x0C）。

表 9 - 29 DMA 编程常用的 DMA 寄存器

名称	端口地址	
	（通道 0-3）	（通道 4-7）
命令寄存器	0x08	0xD0
请求寄存器	0x09	0xD2
单屏蔽寄存器	0x0A	0xD4
方式寄存器	0x0B	0xD6
清除先后触发器	0x0C	0xD8

命令寄存器用于规定 DMA 控制器芯片的操作要求，设定 DMA 控制器的总体状态。通常它在开机初始化之后就无须变动。在 Linux 0.11 内核中，软盘驱动程序就直接使用了开机后 ROM BIOS 的设置值。作为参考，这里列出命令寄存器各比特位的含义，见表 9-30 所示。（在读该端口时，所得内容是 DMA 控制器状态寄存器的信息）

表 9-30 DMA 命令寄存器格式

位	说明
7	DMA 响应外设信号 DACK：0-DACK 低电平有效；1-DACK 高电平有效。
6	外设请求 DMA 信号 DREQ：0-DREQ 低电平有效；1-DREQ 高电平有效。
5	写方式选择：0-选择迟后写；1-选择扩展写；X-若位 3=1。
4	DMA 通道优先方式：0-固定优先；1-轮转优先。
3	DMA 周期选择：0-普通定时周期（5）；1-压缩定时周期（3）；X-若位 0=1。
2	开启 DMA 控制器：0-允许控制器工作；1-禁止控制器工作。
1	通道 0 地址保持：0-禁止通道 0 地址保持；1-允许通道 0 地址保持；X-若位 0=0。
0	内存传输方式：0-禁止内存至内存传输方式；1-允许内存至内存传输方式。

请求寄存器用于记录外设对通道的请求服务信号 DREQ。每个通道对应一位。当 DREQ 有效时对应位置 1，当 DMA 控制器对其作出响应时会对该位置 0。如果不使用 DMA 的请求信号 DREQ 引脚，那么也可以通过编程直接设置相应通道的请求位来请求 DMA 控制器的服务。在 PC 机中，软盘控制器与 DMA 控制器的通道 2 有直接的请求信号 DREQ 连接，因此 Linux 内核中也无须对该寄存器进行操作。作为参考，这里还是列出请求通道服务的字节格式，见表 9-31 所示。

表 9-31 DMA 请求寄存器各比特位的含义

位	说明
7-3	不用。
2	屏蔽标志。0 - 请求位置位；1 - 请求位复位（置 0）。
1	通道选择。00-11 分别选择通道 0-3。
0	

单屏蔽寄存器的端口是 0x0A（对于 16 位通道则是 0xD4）。一个通道被屏蔽，是指使用该通道的外设发出的 DMA 请求信号 DREQ 得不到 DMA 控制器的响应，因此也就无法让 DMA 控制器操作该通道。该寄存器各比特位的含义见表 9-32 所示。

表 9-32 DMA 单屏蔽寄存器各比特位的含义

位	说明
7-3	不用。
2	屏蔽标志。1 - 屏蔽选择的通道；0 - 开启选择的通道。
1	通道选择。00-11 分别选择通道 0-3。
0	

方式寄存器用于指定某个 DMA 通道的操作方式。在 Linux 0.11 内核中，使用了其中读（0x46）和写（0x4A）两种方式。该寄存器各位的含义见表 9-33 所示。

表 9-33 DMA 方式寄存器各比特位的含义

位	说明
7	选择传输方式：00-请求模式；01-单字节模式；10-块字节模式；11-接连模式。
6	
5	地址增减方式。0-地址递减；1-地址递增。
4	自动预置（初始化）。0-自动预置；1-非自动预置。
3	传输类型：00-DMA 校验；01-DMA 读传输；10-DMA 写传输。11-无效。
2	
1	通道选择。00-11 分别选择通道 0-3。
0	

由于通道的地址和计数寄存器可以读写 16 位的数据，因此在设置他们时都需要分别执行两次写操作，一次访问低字节，一次访问高字节。而实际写哪个字节则由先后触发器的状态决定。清除先后触发器端口 0x0C 就是用于在读/写 DMA 控制器中地址或计数信息之前把字节先后触发器初始化为默认状态。当字节触发器为 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，该触发器就变化一次。写 0x0C 端口就可以将触发器置成 0 状态。








在使用 DMA 控制器时，通常需要按照一定的步骤来进行，下面以软盘驱动程序使用 DMA 控制器的方式来加以说明：

1. 关中断，以排除任何干扰；
2. 修改屏蔽寄存器（端口 0x0A），以屏蔽需要使用的 DMA 通道。对于软盘驱动程序来说就是通道 2；
3. 向 0x0C 端口写操作，置字节先后触发器为默认状态；
4. 写方式寄存器（端口 0x0B），以设置指定通道的操作方式字。对于；
5. 写地址寄存器（端口 0x04），设置 DMA 使用的内存页面中的偏移地址。先写低字节，后写高字节；
6. 写页面寄存器（端口 0x81），设置 DMA 使用的内存页面；
7. 写计数寄存器（端口 0x05），设置 DMA 传输的字节数。应该是传输长度-1。同样需要针对高低字节分别写一次。本书中软盘驱动程序每次要求 DMA 控制器传输的长度是 1024 字节，因此写 DMA 控制器的长度值应该是 1023（即 0x3FF）；
8. 再次修改屏蔽寄存器（端口 0x0A），以开启 DMA 通道；
9. 最后，开启中断，以允许软盘控制器在传输结束后向系统发出中断请求。

第10章 字符设备驱动程序(char driver)

在 linux 0.11 内核中, 字符设备主要包括控制终端设备和串行终端设备。本章的代码就是用于对这些设备的输入输出进行操作。有关终端驱动程序的工作原理可参考 M.J.Bach 的《UNIX 操作系统设计》第 10 章第 3 节内容。

列表 10-1 linux/kernel/chr_drv 目录

	文件名	大小	最后修改时间(GMT)	说明
	Makefile	2443 bytes	1991-12-02 03:21:41	Make 配置文件
	console.c	14568 bytes	1991-11-23 18:41:21	控制台处理
	keyboard.S	12780 bytes	1991-12-04 15:07:58	键盘中断处理
	rs_io.s	2718 bytes	1991-10-02 14:16:30	串行中断处理
	serial.c	1406 bytes	1991-11-17 21:49:05	串行初始化
	tty_io.c	7634 bytes	1991-12-08 18:09:15	终端 IO 处理
	tty_ioctl.c	4979 bytes	1991-11-25 19:59:38	终端 IO 控制

10.1 总体功能

本章的程序可分成三部分。第一部分是关于 RS-232 串行线路驱动程序, 包括程序 `rs_io.s` 和 `serial.c`; 另一部分是涉及控制台驱动程序, 这包括键盘中断驱动程序 `keyboard.S` 和控制台显示驱动程序 `console.c`; 第三部分是终端驱动程序与上层接口部分, 包括终端输入输出程序 `tty_io.c` 和终端控制程序 `tty_ioctl.c`。下面我们首先概述终端控制驱动程序实现的基本原理, 然后再分这三部分分别说明它们的基本功能。

10.1.1 终端驱动程序基本原理

终端驱动程序用于控制终端设备, 在终端设备和进程之间传输数据, 并对所传输的数据进行一定的处理。用户在键盘上键入的原始数据 (Raw data), 在通过终端程序处理后, 被传送给一个接收进程; 而进程向终端发送的数据, 在终端程序处理后, 被显示在终端屏幕上或者通过串行线路被发送到远程终端。根据终端程序对待输入或输出数据的方式, 可以把终端工作模式分成两种。一种是规范模式 (canonical), 此时经过终端程序的数据将被进行变换处理, 然后再送出。例如把 TAB 字符扩展为 8 个空格字符, 用键入的删除字符 (backspace) 控制删除前面键入的字符等。使用的处理函数一般称为行规则 (line discipline) 模块。另一种是非规范模式或称原始 (raw) 模式。在这种模式下, 行规则程序仅在终端与进程之间传送数据, 而不对数据进行规范模式的变换处理。

在终端驱动程序中, 根据它们与设备的关系, 以及在执行流程中的位置, 可以分为字符设备的直接驱动程序和与上层直接联系的接口程序。我们可以用图 10-1 示意图来表示这种控制关系。

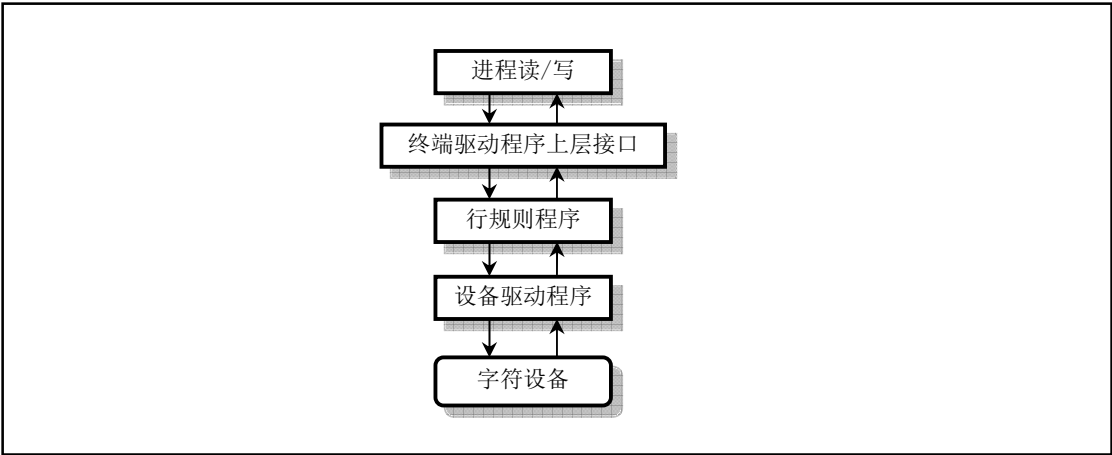


图 10-1 终端驱动程序控制流程

10.1.2 Linux 支持的终端设备类型

终端是一种字符型设备，它有多种类型。我们通常使用 `tty` 来简称各种类型的终端设备。`tty` 是 Teletype 的缩写。Teletype 是一种由 Teletype 公司生产的最早出现的终端设备，样子很象电传打字机。在 Linux 0.1x 系统设备文件目录 `/dev/` 中，通常包含以下一些终端设备文件：

crw-rw-rw-	1 root	tty	5,	0 Jul 30 1992	tty	// 控制终端。
crw--w--w-	1 root	tty	4,	0 Jul 30 1992	tty0	// 当前虚拟终端别名。
crw--w--w-	1 root	tty	4,	1 Jul 30 1992	console	// 控制台。
crw--w--w-	1 root	other	4,	1 Jul 30 1992	tty1	// 虚拟终端 1。
crw--w--w-	1 root	tty	4,	2 Jul 30 1992	tty2	
crw--w--w-	1 root	tty	4,	3 Jul 30 1992	tty3	
crw--w--w-	1 root	tty	4,	4 Jul 30 1992	tty4	
crw--w--w-	1 root	tty	4,	5 Jul 30 1992	tty5	
crw--w--w-	1 root	tty	4,	6 Jul 30 1992	tty6	
crw--w--w-	1 root	tty	4,	7 Jul 30 1992	tty7	
crw--w--w-	1 root	tty	4,	8 Jul 30 1992	tty8	
crw-rw-rw-	1 root	tty	4,	64 Jul 30 1992	ttys1	// 串行端口终端 1。
crw-rw-rw-	1 root	tty	4,	65 Jul 30 1992	ttys2	
crw--w--w-	1 root	tty	4,	128 Jul 30 1992	ptyp0	// 主伪终端。
crw--w--w-	1 root	tty	4,	129 Jul 30 1992	ptyp1	
crw--w--w-	1 root	tty	4,	130 Jul 30 1992	ptyp2	
crw--w--w-	1 root	tty	4,	131 Jul 30 1992	ptyp3	
crw--w--w-	1 root	tty	4,	192 Jul 30 1992	ttyp0	// 从伪终端。
crw--w--w-	1 root	tty	4,	193 Jul 30 1992	ttyp1	
crw--w--w-	1 root	tty	4,	194 Jul 30 1992	ttyp2	
crw--w--w-	1 root	tty	4,	195 Jul 30 1992	ttyp3	

这些终端设备文件可以分为以下几种类型：

1. 串行端口终端（`/dev/ttySn`）

串行端口终端是使用计算机串行端口连接的终端设备。计算机把每个串行端口都看作是一个字符设备。有段时间这些串行端口设备通常被称为终端设备，因为那时它的最大用途就是用来连接终端。这些串行端口所对应的设备文件名是 `/dev/ttyS0`、`/dev/ttyS1` 等，设备号分别是（4,64）、（4,65）等，分别对应于 DOS 系统下的 COM1、COM2。若要向一个端口发送数据，可以在命令行上把标准输出重定向到这些

特殊文件名上即可。例如，在命令行提示符下键入：`echo test > /dev/ttyS1`，就会把单词“test”发送到连接在 `ttyS1` 端口的设备上。

2. 伪终端 (/dev/ptyp、/dev/ttyp)

伪终端 (Pseudo Terminals, 或 Pseudo - TTY, 简称为 PTY) 是一种功能类似于一般终端的设备, 但是这种设备并不与任何终端硬件相关。伪终端设备用于为其它程序提供类似于终端式样的接口, 主要应用于通过网络登录主机时为网络服务器和登录 shell 程序之前提供一个终端接口, 或为运行于 X Window 窗口中的终端程序提供终端样式的接口。当然, 我们也可以利用伪终端在任何两个使用终端接口的程序之间建立数据读写通道。为了分别为两个应用程序或进程提供终端样式接口, 伪终端均配对使用。一个被称为主伪终端 (Master PTY) 或伪终端主设备, 另一个称为从伪终端 (Slave PTY) 或伪终端从设备。对于象 `ptyp1` 和 `ttyp1` 这样成对的伪终端逻辑设备来讲, `ptyp1` 是主设备或者是控制终端, 而 `ttyp1` 则是从设备。往其中任意一个伪终端写入的数据会通过内核直接由配对的伪终端接收到。例如对于主设备 `/dev/ptyp3` 和从设备 `/dev/ttyp3`, 如果一个程序把 `ttyp3` 看作是一个串行端口设备, 则它对该端口的读/写操作会反映在对应的另一个逻辑终端设备 `ptyp3` 上面。而 `ptyp3` 则会是另一个程序用于读写操作的逻辑设备。这样, 两个程序就可以通过这种逻辑设备进行互相交流, 而其中一个使用从设备 `ttyp3` 的程序则认为自己正在与一个串行端口进行通信。这很像是逻辑设备对之间的管道操作。对于伪终端从设备, 任何一个设计成使用串行端口设备的程序都可以使用该逻辑设备。但对于使用主设备的程序来讲, 就需要专门设计来使用伪终端主设备。

例如, 如果某人在网上使用 `telnet` 程序连接到你的计算机上, 那么 `telnet` 程序就可能会开始连接到伪终端主设备 `ptyp2` 上。此时一个 `getty` 程序就应该运行在对应的 `ttyp2` 端口上。当 `telnet` 从远端获取了一个字符时, 该字符就会通过 `ptyp2`、`ttyp2` 传递给 `getty` 程序, 而 `getty` 程序则会通过 `ttyp2`、`ptyp2` 和 `telnet` 程序往网络上送出“login:”字符串信息。这样, 登录程序与 `telnet` 程序就通过“伪终端”进行通信。通过使用适当的软件, 我们就可以把两个甚至多个伪终端设备连接到同一个物理端口上。

以前的 Linux 系统最多只有 16 个成对的 `ttyp` (`ttyp0—ttypf`) 设备文件名。但现在的 Linux 系统上通常都使用“主伪终端 (ptm - pty master)”命名方式, 例如 `/dev/ptm3`。它的对应端则会被自动创建成 `/dev/pts/3`。这样就可以在需要时动态提供一个 `pty` 伪终端。现在的 Linux 系统上目录 `/dev/pts` 是一个 `devpts` 类型的文件系统。虽然“文件”`/dev/pts/3` 看上去是设备文件系统中的一项, 但其实它完全是一种不同的文件系统。

3. 控制终端 (/dev/tty)

字符设备文件 `/dev/tty` 是进程控制终端 (Controlling Terminal) 的别名, 其主设备号是 5, 次设备号是 0。如果当前进程有控制终端, 那么 `/dev/tty` 就是当前进程控制终端的设备文件。我们可以使用命令“`ps -ax`”来查看进程与哪个控制终端相连。对于登录 shell 来讲, `/dev/tty` 就是我们使用的终端, 其设备号是 (5,0)。我们可以使用命令“`tty`”来查看它具体对应哪个实际终端设备。实际上 `/dev/tty` 有些类似于连接到实际终端设备的一个链接。

如果一个终端用户执行了一个程序, 但不想要控制终端 (例如一个后台服务器程序), 那么进程可以先试着打开 `/dev/tty` 文件。如果打开成功, 则说明进程有控制终端。此时我们可以使用 `TIOCNOTTY` (Terminal IO Control NO TTY) 参数的 `ioctl()` 调用来放弃控制终端。

4. 控制台 (/dev/ttyn, /dev/console)

在 Linux 系统中, 计算机显示器通常被称为控制台终端或控制台 (Console)。它仿真了 VT200 或 Linux 类型终端 (TERM=Linux), 并且有一些字符设备文件与之关联: `tty0`、`tty1`、`tty2` 等。当我们在控制台上登录时, 使用的就是 `tty1`。另外, 使用 `Alt+[F1—F6]` 组合键我们就可以切换到 `tty2`、`tty3` 等上面去。`tty1—tty6` 被称为虚拟终端, 而 `tty0` 则是当前所使用虚拟终端的一个别名。Linux 系统所产生的信息都会发送到 `tty0` 上。因此不管当前正在使用哪个虚拟终端, 系统信息都会发送到我们的屏幕上。你可以登录到不同的虚拟终端上去, 因而可以让系统同时有几个不同的会话存在。但只有系统或超级用户 `root` 可以向 `/dev/tty0` 执行写操作。而且有时 `/dev/console` 也会连接至该终端设备上。但在 Linux 0.12 系统中, `/dev/console` 通常连接到第 1 个虚拟终端 `tty1` 上。

5. 其它类型

现在的 Linux 系统中还针对很多不同的字符设备建有很多其它种类的终端设备特殊文件。例如针对 ISDN 设备的/dev/ttyIn 终端设备等。这里不再赘述。

10.1.3 终端基本数据结构

每个终端设备都对应有一个 `tty_struct` 数据结构，主要用来保存终端设备当前参数设置、所属的前台进程组 ID 和字符 IO 缓冲队列等信息。该结构定义在 `include/linux/tty.h` 文件中，其结构如下所示：

```
struct tty\_struct {
    struct termios termios;           // 终端 io 属性和控制字符数据结构。
    int pgrp;                         // 所属进程组。
    int stopped;                      // 停止标志。
    void (*write)(struct tty\_struct * tty); // tty 写函数指针。
    struct tty\_queue read_q;          // tty 读队列。
    struct tty\_queue write_q;         // tty 写队列。
    struct tty\_queue secondary;       // tty 辅助队列(存放规范模式字符序列)，
};                                   // 可称为规范(熟)模式队列。
extern struct tty\_struct tty\_table[]; // tty 结构数组。
```

Linux 内核使用了数组 `tty_table[]` 来保存系统中每个终端设备的信息。每个数组项是一个数据结构 `tty_struct`，对应系统中一个终端设备。Linux 0.11 内核共支持三个终端设备。一个是控制台设备，另外两个是使用系统上两个串行端口的串行终端设备。

`termios` 结构用于存放对应终端设备的 io 属性。有关该结构的详细描述见下面说明。`pgrp` 是进程组标识，它指明一个会话中处于前台的进程组，即当前拥有该终端设备的进程组。`pgrp` 主要用于进程的作业控制操作。`stopped` 是一个标志，表示对应终端设备是否已经停止使用。函数指针 `*write()` 是该终端设备的输出处理函数，对于控制台终端，它负责驱动显示硬件，在屏幕上显示字符等信息。对于通过系统串行端口连接的串行终端，它负责把输出字符发送到串行端口。

终端所处理的数据被保存在 3 个 `tty_queue` 结构的字符缓冲队列中（或称为字符表），见下面所示：

```
struct tty\_queue {
    unsigned long data;           // 等待队列缓冲区中当前数据统计值。
                                   // 对于串口终端，则存放串口端口地址。
    unsigned long head;           // 缓冲区中数据头指针。
    unsigned long tail;          // 缓冲区中数据尾指针。
    struct task\_struct * proc_list; // 等待本缓冲队列的进程列表。
    char buf[1024];              // 队列的缓冲区。
};
```

每个字符缓冲队列的长度是 1K 字节。其中读缓冲队列 `read_q` 用于临时存放从键盘或串行终端输入的原始（raw）字符序列；写缓冲队列 `write_q` 用于存放写到控制台显示屏或串行终端去的数据；根据 ICANON 标志，辅助队列 `secondary` 用于存放从 `read_q` 中取出的经过行规则程序处理（过滤）过的数据，或称为熟(cooked)模式数据。这是在行规则程序把原始数据中的特殊字符如删除（backspace）字符变换后的规范输入数据，以字符行为单位供应用程序读取使用。上层终端读函数 `tty_read()` 即用于读取 `secondary` 队列中的字符。

在读入用户键入的数据时，中断处理汇编程序只负责把原始字符数据放入输入缓冲队列中，而由中断处理过程中调用的 C 函数（`copy_to_cooked()`）来处理字符的变换工作。例如当进程向一个终端写数据时，终端驱动程序就会调用行规则函数 `copy_to_cooked()`，把用户缓冲区中的所有数据数据到写缓冲队

列中，并将数据发送到终端上显示。在终端上按下一个键时，所引发的键盘中断处理过程会把按键扫描码对应的字符放入读队列 `read_q` 中，并调用规范模式处理程序把 `read_q` 中的字符经过处理再放入辅助队列 `secondary` 中。与此同时，如果终端设备设置了回显标志 (`L_ECHO`)，则也把该字符放入写队列 `write_q` 中，并调用终端写函数把该字符显示在屏幕上。通常除了象键入密码或其他特殊要求以外，回显标志都是置位的。我们可以通过修改终端的 `termios` 结构中的信息来改变这些标志值。

在上述 `tty_struct` 结构中还包括一个 `termios` 结构，该结构定义在 `include/termios.h` 头文件中，其字段内容如下所示：

```
struct termios {
    unsigned long c_iflag;           /* input mode flags */      // 输入模式标志。
    unsigned long c_oflag;           /* output mode flags */     // 输出模式标志。
    unsigned long c_cflag;           /* control mode flags */    // 控制模式标志。
    unsigned long c_lflag;           /* local mode flags */      // 本地模式标志。
    unsigned char c_line;             /* line discipline */       // 线路规程（速率）。
    unsigned char c_cc[NCCS];        /* control characters */    // 控制字符数组。
};
```

其中，`c_iflag` 是输入模式标志集。Linux 0.11 内核实现了 POSIX.1 定义的所有 11 个输入标志，参见 `termios.h` 头文件中的说明。终端设备驱动程序用这些标志来控制如何对终端输入的字符进行变换（过滤）处理。例如是否需要把输入的换行符（NL）转换成回车符（CR）、是否需要把输入的大写字符转换成小写字符（因为以前有些终端设备只能输入大写字符）等。在 Linux 0.11 内核中，相关的处理函数是 `tty_io.c` 文件中的 `copy_to_cooked()`。参见 `termios.h` 文件第 83 -- 96 行。

`c_oflag` 是输出模式标志集。终端设备驱动程序使用这些标志控制如何把字符输出到终端上，主要在 `tty_io.c` 的 `tty_write()` 函数中使用。参见 `termios.h` 文件第 99 -- 129 行。

`c_cflag` 是控制模式标志集。主要用于定义串行终端传输特性，包括波特率、字符比特位数以及停止位数等。参见 `termios.h` 文件中第 132 -- 166 行。

`c_lflag` 是本地模式标志集。主要用于控制驱动程序与用户的交互。例如是否需要回显（Echo）字符、是否需要把擦除字符直接显示在屏幕上、是否需要让终端上键入的控制字符产生信号。这些操作主要在 `copy_to_cooked()` 函数和 `tty_read()` 中使用。例如，若设置了 `ICANON` 标志，则表示终端处于规范模式输入状态，否则终端处于非规范模式。如果设置 `ISIG` 标志，则表示收到终端发出的控制字符 `INTR`、`QUIT`、`SUSP` 时系统需要产生相应的信号。参见 `termios.h` 文件中第 169 -- 183 行。

上述 4 种标志集的类型都是 `unsigned long`，每个比特位可表示一种标志，因此每个标志集最多可有 32 个输入标志。所有这些标志及其含义可参见 `termios.h` 头文件。

`c_cc[]` 数组包含了终端所有可以修改的特殊字符。例如你可以通过修改其中的中断字符（`^C`）由其他按键产生。其中 `NCCS` 是数组的长度值。终端默认的 `c_cc[]` 数组初始值定义在 `include/linux/tty.h` 文件中。程序引用该数组中各项时定义了数组项符号名，这些名称都以字母 `V` 开头，例如 `VINTR`、`VMIN`。参见 `termios.h` 第 64 -- 80 行。

因此，利用系统调用 `ioctl` 或使用相关函数 (`tcsetattr()`)，我们可以通过修改 `termios` 结构中的信息来改变终端的设置参数。行规则函数即是根据这些设置参数进行操作。例如，控制终端是否要对键入的字符进行回显、设置串行终端传输的波特率、清空读缓冲队列和写缓冲队列。

当用户修改终端参数，将规范模式标志复位，则就会把终端设置为工作在原始模式，此时行规则程序会把用户键入的数据原封不动地传送给用户，而回车符也被当作普通字符处理。因此，在用户使用系统调用 `read` 时，就应该作出某种决策方案以判断系统调用 `read` 什么时候算完成并返回。这将由终端 `termios` 结构中的 `VTIME` 和 `VMIN` 控制字符决定。这两个是读操作的超时定时值。`VMIN` 表示为了满足

读操作，需要读取的最少字符数；VTIME 则是一个读操作等待定时值。

我们可以使用命令 stty 来查看当前终端设备 termios 结构中标志的设置情况。在 Linux 0.1x 系统命令行提示符下键入 stty 命令会显示以下信息：

```
[/root]# stty
-----Characters-----
INTR:  '^C'  QUIT:  '^\'  ERASE:  '^H'  KILL:   '^U'  EOF:    '^D'
TIME:   0    MIN:    1   SWTC:   '^@'  START:  '^Q'  STOP:   '^S'
SUSP:   '^Z'  EOL:    '^@'  EOL2:   '^@'  LNEXT:  '^V'
DISCARD: '^O'  REPRINT: '^R'  RWERASE: '^W'
-----Control Flags-----
-CSTOPB  CREAD -PARENB -PARODD  HUPCL -CLOCAL -CRTSCTS
Baud rate: 9600 Bits: CS8
-----Input Flags-----
-IGNBRK -BRKINT -IGNPAR -PARMRK -INPCK -ISTRIP -INLCR -IGNCR
ICRNL -IUCLC  IXON  -IXANY  IXOFF -IMAXBEL
-----Output Flags-----
OPOST -OLCUC  ONLCR -OCRNL -ONOCR -ONLRET -OFILL -OFDEL
Delay modes: CR0 NL0 TAB0 BSO FFO VTO
-----Local Flags-----
ISIG ICANON -XCASE ECHO -ECHOE -ECHOK -ECHONL -NOFLSH
-TOSTOP ECHOCTL ECHOPRT ECHOKE -FLUSHO -PENDIN -IEXTEN
rows 0 cols 0
```

其中带有减号标志表示没有设置。另外对于现在的 Linux 系统，需要键入'stty -a'才能显示所有这些信息，并且显示格式有所区别。

终端程序所使用的上述主要数据结构和它们之间的关系可见图 10-2 所示。

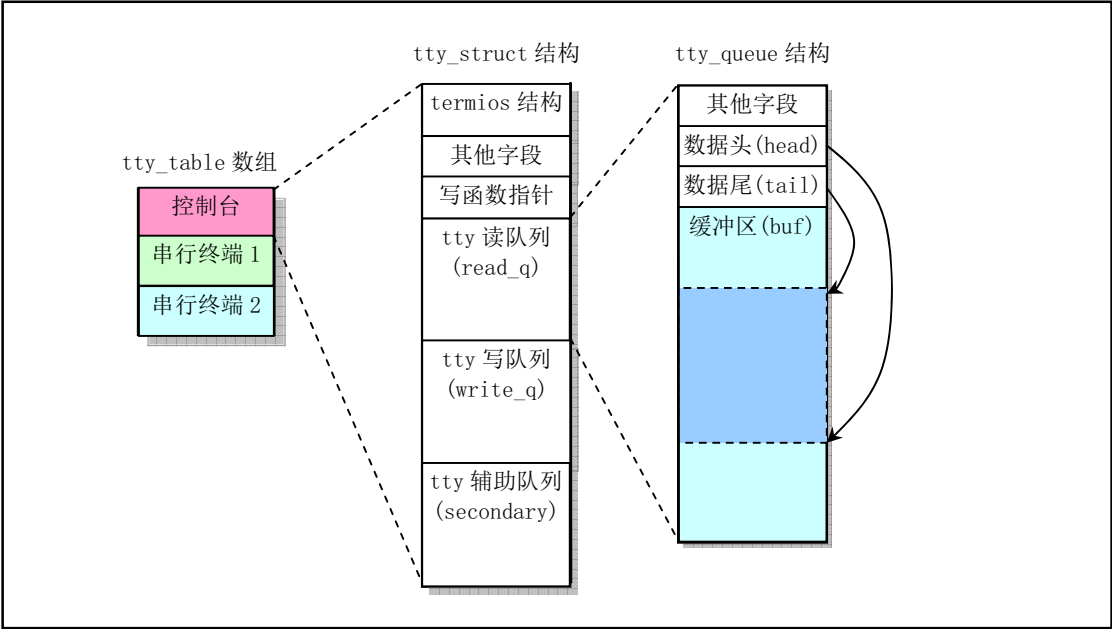


图 10-2 终端程序的数据结构

10.1.4 规范模式和非规范模式

10.1.4.1 规范模式

当 `c_lflag` 中的 `ICANON` 标志置位时，则按照规范模式对终端输入数据进行处理。此时输入字符被装配成行，进程以字符行的形式读取。当一行字符输入后，终端驱动程序会立刻返回。行的定界符有 `NL`、`EOL`、`EOL2` 和 `EOF`。其中除最后一个 `EOF`（文件结束）将被处理程序删除外，其余四个字符将被作为一行的最后一个字符返回给调用程序。

在规范模式下，终端输入的以下字符将被处理：`ERASE`、`KILL`、`EOF`、`EOL`、`REPRINT`、`WERASE` 和 `EOL2`。

`ERASE` 是擦除字符（Backspace）。在规范模式下，当 `copy_to_cooked()` 函数遇该输入字符时会删除缓冲队列中最后输入的一个字符。若队列中最后一个字符是上一行的字符（例如是 `NL`），则不作任何处理。此后该字符被忽略，不放到缓冲队列中。

`KILL` 是删行字符。它删除队列中最后一行字符。此后该字符被忽略掉。

`EOF` 是文件结束符。在 `copy_to_cooked()` 函数中该字符以及行结束字符 `EOL` 和 `EOL2` 都将被当作回车符来处理。在读操作函数中遇到该字符将立即返回。`EOF` 字符不会放入队列中而是被忽略掉。

`REPRINT` 和 `WERASE` 是扩展规范模式下识别的字符。`REPRINT` 会让所有未读的输入被输出。而 `WERASE` 用于擦除单词（跳过空白字符）。在 `Linux 0.11` 中，程序忽略了对这两个字符的识别和处理。

10.1.4.2 非规范模式

如果 `ICANON` 处于复位状态，则终端程序工作在非规范模式下。此时终端程序不对上述字符进行处理，而是将它们当作普通字符处理。输入数据也没有行的概念。终端程序何时返回读进程是由 `MIN` 和 `TIME` 的值确定。这两个变量是 `c_cc[]` 数组中的变量。通过修改它们即可改变在非规范模式下进程读字符的处理方式。

`MIN` 指明读操作最少需要读取的字符数；`TIME` 指定等待读取字符的超时值（计量单位是 1/10 秒）。根据它们的值可分四种情况来说明。

1. `MIN>0`, `TIME>0`

此时 `TIME` 是一个字符间隔超时定时值，在接收到第一个字符后才起作用。在超时之前，若先接收到了 `MIN` 个字符，则读操作立刻返回。若在收到 `MIN` 个字符之前超时了，则读操作返回已经接收到的字符数。此时起码能返回一个字符。因此在接收到一个字符之前若 `secondary` 空，则读进程将被阻塞（睡眠）。

2. `MIN>0`, `TIME=0`

此时只有在收到 `MIN` 个字符时读操作才返回。否则就无限期待（阻塞）。

3. `MIN=0`, `TIME>0`

此时 `TIME` 是一个读操作超时定时值。当收到一个字符或者已超时，则读操作就立刻返回。如果是超时返回，则读操作返回 0 个字符。

4. `MIN=0`, `TIME=0`

在这种设置下，如果队列中有数据可以读取，则读操作读取需要的字符数。否则立刻返回 0 个字符数。

在以上四种情况中，`MIN` 仅表明最少读到的字符数。如果进程要求读取比 `MIN` 要多的字符，那么只要队列中有就可能满足进程的当前需求。有关对终端设备的读操作处理，请参见程序 `tty_io.c` 中的 `tty_read()` 函数。

10.1.5 控制台终端和串行终端设备

在 `Linux 0.11` 系统中可以使用两类终端。一类是主机上的控制台终端，另一类是串行硬件终端设备。控制台终端由内核中的键盘中断处理程序 `keyboard.s` 和显示控制程序 `console.c` 进行管理。它接收上层

`tty_io.c` 程序传递下来的显示字符或控制信息，并控制在主机屏幕上字符的显示，同时控制台（主机）把键盘按键产生的代码经由 `keyboard.s` 传送到 `tty_io.c` 程序去处理。串行终端设备则通过线路连接到计算机串行端口上，并通过内核中的串程序 `rs_io.s` 与 `tty_io.c` 直接进行信息交互。

`keyboard.s` 和 `console.c` 这两个程序实际上是 Linux 系统主机中使用显示器和键盘模拟一个硬件终端设备的仿真程序。只是由于在主机上，因此我们称这个模拟终端环境为控制台终端，或直接称为控制台。这两个程序所实现的功能就相当于一个串行终端设备固化在 ROM 中的终端处理程序的作用（除了通信部分），也象普通 PC 机上的一个终端仿真软件。因此虽然程序在内核中，但我们还是可以独立地看待它们。这个模拟终端与普通的硬件终端设备主要的区别在于不需要通过串行线路通信驱动程序。因此 `keyboard.s` 和 `console.c` 程序必须模拟一个实际终端设备（例如 DEC 的 VT100 终端）具备的所有硬件处理功能，即终端设备固化程序中除通信以外的所有处理功能。控制台终端和串行终端设备在处理结构上的相互区别与类似之处参见图 10-3。所以如果我们对一般硬件终端设备或终端仿真程序工作原理有一定了解，那么阅读这两个程序就不会碰到什么困难。

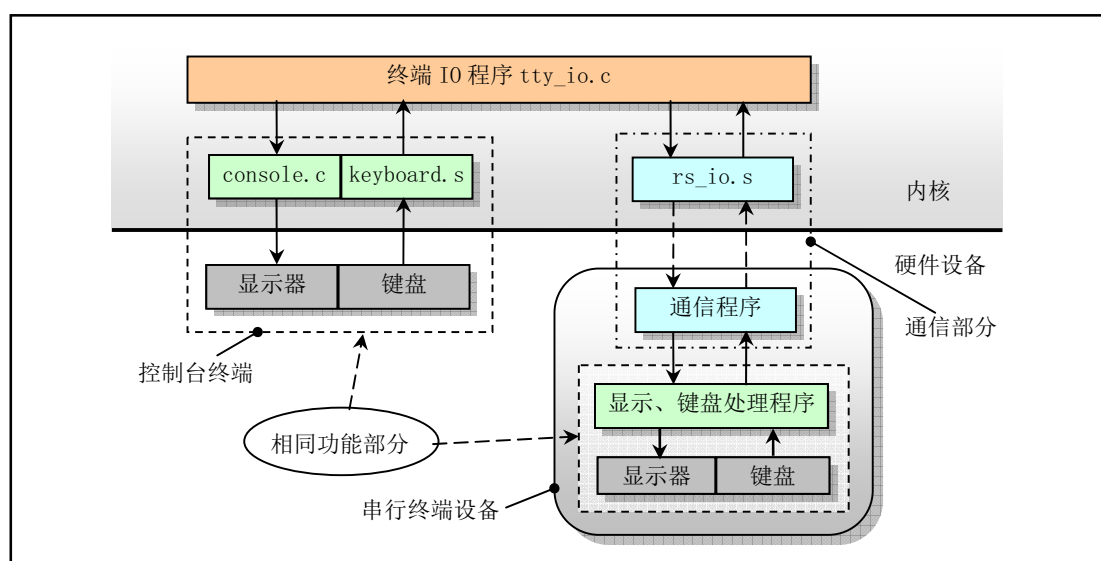
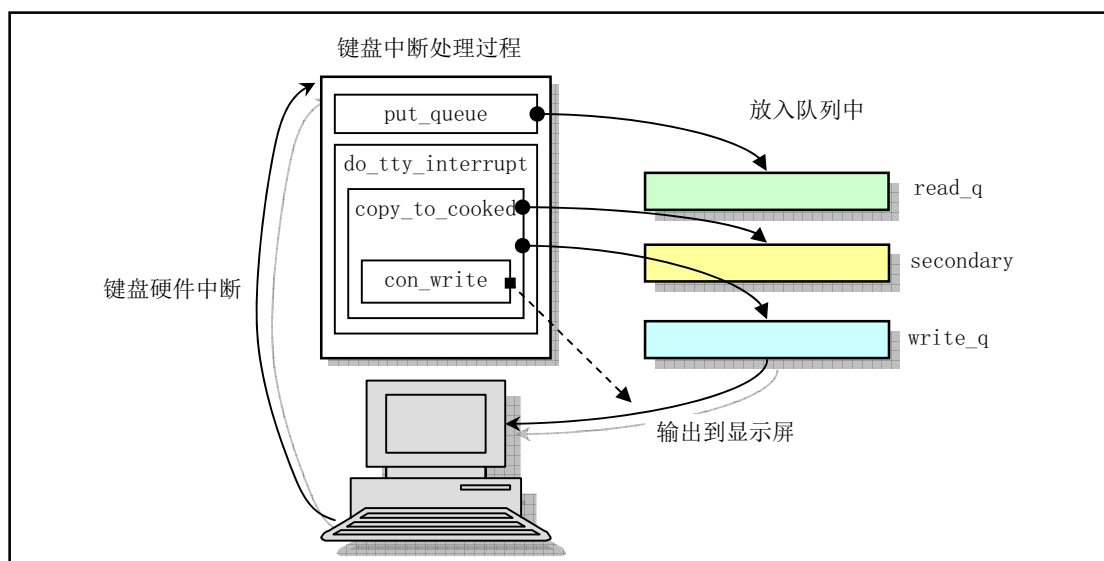


图 10-3 控制台终端与串行终端设备示意图

10.1.5.1 控制台驱动程序

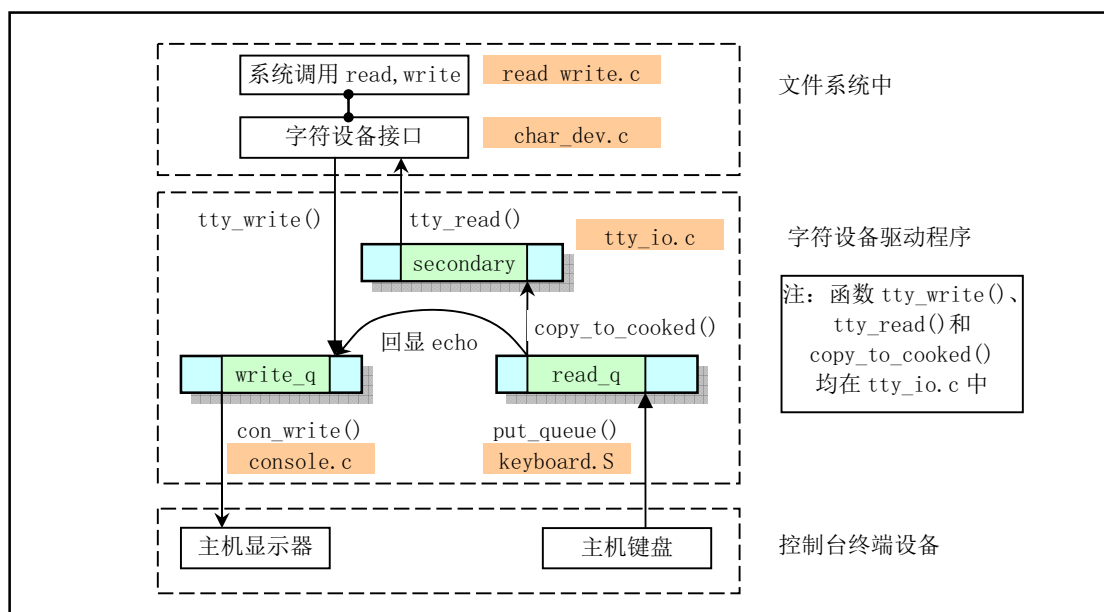
在 Linux 0.11 内核中，终端控制台驱动程序涉及 `keyboard.S` 和 `console.c` 程序。`keyboard.S` 用于处理用户键入的字符，把它们放入读缓冲队列 `read_q` 中，并调用 `copy_to_cooked()` 函数读取 `read_q` 中的字符，经转换后放入辅助缓冲队列 `secondary`。`console.c` 程序实现控制台终端收到代码的输出处理。

例如，当用户在键盘上键入了一个字符时，会引起键盘中断响应（中断请求信号 `IRQ1`，对应中断号 `INT 33`），此时键盘中断处理程序就会从键盘控制器读入对应的键盘扫描码，然后根据使用的键盘扫描码映射表译成相应字符，放入 `tty` 读队列 `read_q` 中。然后调用中断处理程序的 C 函数 `do_tty_interrupt()`，它又直接调用行规则函数 `copy_to_cooked()` 对该字符进行过滤处理，并放入 `tty` 辅助队列 `secondary` 中，同时把该字符放入 `tty` 写队列 `write_q` 中，并调用写控制台函数 `con_write()`。此时如果该终端的回显（`echo`）属性是设置的，则该字符会显示到屏幕上。`do_tty_interrupt()` 和 `copy_to_cooked()` 函数在 `tty_io.c` 中实现。整个操作过程见图 10-4 所示。



对于进程执行 `tty` 写操作，终端驱动程序是一个字符一个字符进行处理的。在写缓冲队列 `write_q` 没有满时，就从用户缓冲区取一个字符，经过处理放入 `write_q` 中。当把用户数据全部放入 `write_q` 队列或者此时 `write_q` 已满，就调用终端结构 `tty_struct` 中指定的写函数，把 `write_q` 缓冲队列中的数据输出到控制台。对于控制台终端，其写函数是 `con_write()`，在 `console.c` 程序中实现。

有关控制台终端操作的驱动程序，主要涉及两个程序。一个是键盘中断处理程序 `keyboard.S`，主要用于把用户键入的字符放入 `read_q` 缓冲队列中；另一个是屏幕显示处理程序 `console.c`，用于从 `write_q` 队列中取出字符并显示在屏幕上。所有这三个字符缓冲队列与上述函数或文件的关系都可以用图 10-5 清晰地表示出来。



10.1.5.2 串行终端驱动程序

处理串行终端操作的程序有 `serial.c` 和 `rs_io.s`。`serial.c` 程序负责对串行端口进行初始化操作。另外，通过取消对发送保持寄存器空中断允许的屏蔽来开启串行中断发送字符操作。`rs_io.s` 程序是串行中断处理过程。主要根据引发中断的 4 种原因分别进行处理。

引起系统发生串行中断的情况有：a. 由于 `modem` 状态发生了变化；b. 由于线路状态发生了变化；c. 由于接收到字符；d. 由于在中断允许标志寄存器中设置了发送保持寄存器中断允许标志，需要发送字符。对引起中断的前两种情况的处理过程是通过读取对应状态寄存器值，从而使其复位。对于由于接收到字符的情况，程序首先把该字符放入读缓冲队列 `read_q` 中，然后调用 `copy_to_cooked()` 函数转换成以字符行为单位的规范模式字符放入辅助队列 `secondary` 中。对于需要发送字符的情况，则程序首先从写缓冲队列 `write_q` 尾指针处取出一个字符发送出去，再判断写缓冲队列是否已空，若还有字符则循环执行发送操作。

对于通过系统串行端口接入的终端，除了需要与控制台类似的处理外，还需要进行串行通信的输入/输出处理操作。数据的读入是由串行中断处理程序放入读队列 `read_q` 中，随后执行与控制台终端一样的操作。

例如，对于一个接在串行端口 1 上的终端，键入的字符将首先通过串行线路传送到主机，引起主机串行口 1 中断请求。此时串行口中断处理程序就会将字符放入串行终端 1 的 tty 读队列 `read_q` 中，然后调用中断处理程序的 C 函数 `do_tty_interrupt()`，它又直接调用行规则函数 `copy_to_cooked()` 对该字符进行过滤处理，并放入 tty 辅助队列 `secondary` 中，同时把该字符放入 tty 写队列 `write_q` 中，并调用写串行终端 1 的函数 `rs_write()`。该函数又会把字符回送给串行终端，此时如果该终端的回显（echo）属性是设置的，则该字符会显示在串行终端的屏幕上。

当进程需要写数据到一个串行终端上时，操作过程与写终端类似，只是此时终端的 `tty_struct` 数据结构中的写函数是串行终端写函数 `rs_write()`。该函数取消对发送保持寄存器空允许中断的屏蔽，从而在发送保持寄存器为空时就会引起串行中断发生。而该串行中断过程则根据此次引起中断的原因，从 `write_q` 写缓冲队列中取出一个字符并放入发送保持寄存器中进行字符发送操作。该操作过程也是一次中断发送一个字符，到最后 `write_q` 为空时就会再次屏蔽发送保持寄存器空允许中断位，从而禁止此类中断发生。

串行终端的写函数 `rs_write()` 在 `serial.c` 程序中实现。串行中断程序在 `rs_io.s` 中实现。串行终端三个字符缓冲队列与函数、程序的关系参见图 10-6 所示。

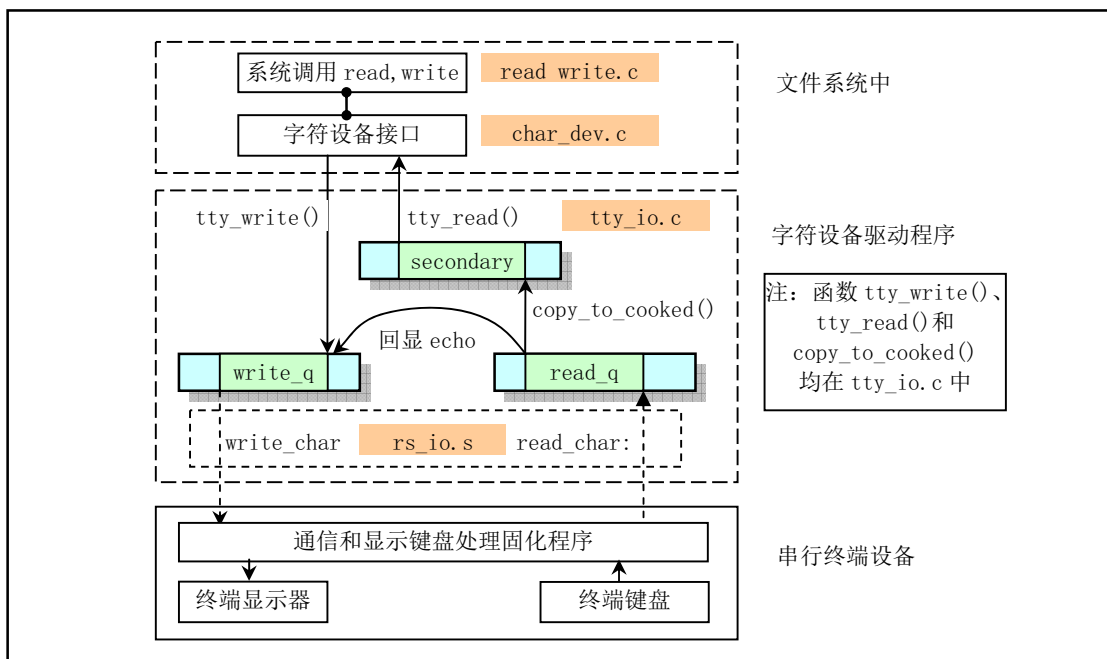


图 10-6 串行终端设备字符缓冲队列与函数之间的关系

由上图可见，串行终端与控制台处理过程之间的主要区别是串行终端利用程序 `rs_io.s` 取代了控制台操作显示器和键盘的程序 `console.c` 和 `keyboard.S`，其余部分的处理过程完全一样。

10.1.6 终端驱动程序接口

通常，用户通过文件系统与设备打交道。每个设备都有一个文件名称，并相应地也在文件系统中占用一个索引节点（i 节点）。但该 i 节点中的文件类型是设备类型，以便与其他正规文件相区别。用户就可以直接使用文件系统调用来访问设备。终端驱动程序也同样为此目的向文件系统提供了调用接口函数。终端驱动程序与系统其他程序的接口是使用 `tty_io.c` 文件中的通用函数实现的。其中实现了读终端函数 `tty_read()` 和写终端函数 `tty_write()`，以及输入行规则函数 `copy_to_cooked()`。另外，在 `tty_ioctl.c` 程序中，实现了修改终端参数的输入输出控制函数（或系统调用）`tty_ioctl()`。终端的设置参数是放在终端数据结构中的 `termios` 结构中，其中的参数比较多，也比较复杂，请参考 `include/termios.h` 文件中的说明。

对于不同终端设备，可以有不同的行规则程序与之匹配。但在 Linux 0.11 中仅有一个行规则函数，因此 `termios` 结构中的行规则字段 `c_line` 不起作用，都被设置为 0。

10.2 Makefile 文件

10.2.1 功能描述

字符设备驱动程序的编译管理程序。由 Make 工具软件使用。

10.2.2 代码注释

程序 10-1 linux/kernel/chr_drv/Makefile

```

1 #
2 # Makefile for the FREAX-kernel character device drivers.
3 #
4 # Note! Dependencies are done automatically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
8 # FREAX(Linux)内核字符设备驱动程序的 Makefile 文件。
9 # 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
10 # 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c文件的信息）。
11
12 AR      =gar      # 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。
13 AS      =gas      # GNU 的汇编程序。
14 LD      =gld      # GNU 的连接程序。
15 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
16 CC      =gcc      # GNU C 语言编译器。
17 # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
18 # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
19 # 的框架指针；-fcombine-reg 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
20 # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己填加的优化选项，以后不再使用；
21 # -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(../../include)。
22 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-reg \

```

```

15      -finline-functions -mstring-insns -nostdinc -I../include
# C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
# 出设备或指定的输出文件中；-nostdinc -I../include 同前。
16 CPP      =gcc -E -nostdinc -I../include
17
# 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
# 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
# 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
# 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。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；若目标是.o，而源文件是.s，则使用第 2 个规则；若目标是.o 文件而源文件
# 是.c 文件，则可直接使用第 3 个规则。
18 .c.s:
19      $(CC) $(CFLAGS) \
20      -S -o $.s $<
# 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22      $(AS) -c -o $.o $<
23 .c.o:                                # 类似上面，*.c 文件→*.o 目标文件。不进行连接。
24      $(CC) $(CFLAGS) \
25      -c -o $.o $<
26
27 OBJS = tty_io.o console.o keyboard.o serial.o rs_io.o \      # 定义目标文件变量 OBJS。
28      tty_ioctl.o
29
# 在有了先决条件 OBJS 后使用下面的命令连接成目标 chr_drv.a 库文件。
# 命令行中的 'rsc' 是操作码和修饰标志（前面可加上 '-'），放置次序任意。其中 'r' 是操作码，
# 说明需要执行的操作。'r' 表示要把命令行末列出的目标文件插入（替换 replacement）归档文件
# blk_drv.a 中。'cs' 是两个修饰标志，用于修饰具体操作行为。'c' 表示当归档文件 blk_drv.a 不
# 存在时就创建这个文件。's' 表示写进或更新归档文件中的目标文件索引。对一个归档文件单独
# 使用命令 "ar s" 等同于对一个归档文件执行命令 ranlib。
30 chr_drv.a: $(OBJS)
31      $(AR) rcs chr_drv.a $(OBJS)
32      sync
33
# 对 keyboard.S 汇编程序进行预处理。-traditional 选项用来对程序作修改使其支持传统的 C 编译器。
# 处理后的程序改名为 kernboard.s。
34 keyboard.s: keyboard.S ../include/linux/config.h
35      $(CPP) -traditional keyboard.S -o keyboard.s
36
# 下面的规则用于清理工作。当执行'make clean'时，就会执行下面的命令，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。
37 clean:
38      rm -f core *.o *.a tmp_make keyboard.s
39      for i in *.c;do rm -f `basename $$i .c`.s;done
40
# 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
# 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
# 文件中'### Dependencies' 行后面的所有行（下面从 48 开始的行），并生成 tmp_make
# 临时文件（44 行的作用）。然后对 kernel/chr_drv/目录下的每个 C 文件执行 gcc 预处理操作。
# -M 标志告诉预处理程序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，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标

```

文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。

```

41 dep:
42     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
43     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,`" "; \
44         $(CPP) -M $$i;done) >> tmp_make
45     cp tmp_make Makefile
46
47 ### Dependencies:
48 console.s console.o : console.c ../../include/linux/sched.h \
49     ../../include/linux/head.h ../../include/linux/fs.h \
50     ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
51     ../../include/linux/tty.h ../../include/termios.h ../../include/asm/io.h \
52     ../../include/asm/system.h
53 serial.s serial.o : serial.c ../../include/linux/tty.h ../../include/termios.h \
54     ../../include/linux/sched.h ../../include/linux/head.h \
55     ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
56     ../../include/signal.h ../../include/asm/system.h ../../include/asm/io.h
57 tty_io.s tty_io.o : tty_io.c ../../include/ctype.h ../../include/errno.h \
58     ../../include/signal.h ../../include/sys/types.h \
59     ../../include/linux/sched.h ../../include/linux/head.h \
60     ../../include/linux/fs.h ../../include/linux/mm.h ../../include/linux/tty.h \
61     ../../include/termios.h ../../include/asm/segment.h \
62     ../../include/asm/system.h
63 tty_ioctl.s tty_ioctl.o : tty_ioctl.c ../../include/errno.h ../../include/termios.h \
64     ../../include/linux/sched.h ../../include/linux/head.h \
65     ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
66     ../../include/signal.h ../../include/linux/kernel.h \
67     ../../include/linux/tty.h ../../include/asm/io.h \
68     ../../include/asm/segment.h ../../include/asm/system.h

```

10.3 keyboard.S 程序

10.3.1 功能描述

该键盘驱动汇编程序主要包括键盘中断处理程序。在英文惯用法中，**make** 表示键被按下；**break** 表示键被松开(放开)。

该程序首先根据键盘特殊键（例如 Alt、Shift、Ctrl、Caps 键）的状态设置程序后面要用到的状态标志变量 **mode** 的值，然后根据引起键盘中断的按键扫描码，调用已经编排成跳转表的相应扫描码处理子程序，把扫描码对应的字符放入读字符队列(**read_q**)中。接下来调用 C 处理函数 **do_tty_interrupt()**(**tty_io.c**, 342 行)，该函数仅包含一个对行规程函数 **copy_to_cooked()** 的调用。这个行规程函数的主要作用就是把 **read_q** 读缓冲队列中的字符经过适当处理后放入规范模式队列（辅助队列 **secondary**）中，并且在处理过程中，若相应终端设备设置了回显标志，还会把字符直接放入写队列（**write_q**）中，从而在终端屏幕上会显示出刚键入的字符。

对于 AT 键盘的扫描码，当键被按下时，则键的扫描码被送出，但当键松开时，将会发送两个字节，第一个是 0xf0，第 2 个还是按下时的扫描码。为了向下的兼容性，设计人员将 AT 键盘发出的扫描码转换成了老式 PC/XT 标准键盘的扫描码。因此这里仅对 PC/XT 的扫描码进行处理即可。有关键盘扫描码

的说明，请参见程序列表后的描述。

另外，这个程序的文件名与其他 `gas` 汇编语言程序不同，它的后缀是大写的 `.S`。使用这样的后缀可以让 `as` 使用 GNU C 编译器的预处理程序 `CPP`，即在你的汇编语言程序中可以使用很多 C 语言的伪指令。例如 `"#include"`、`"#if"` 等，参见程序中的具体使用方法。

10.3.2 代码注释

程序 10-2 linux/kernel/chr_drv/keyboard.S

```

1  /*
2  *  linux/kernel/keyboard.S
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *      Thanks to Alfred Leung for US keyboard patches
9  *              Wolfgang Thiel for German keyboard patches
10 *              Marc Corsini for the French keyboard
11 */
12 /*
13 * 感谢 Alfred Leung 添加了 US 键盘补丁程序；
14 *      Wolfgang Thiel 添加了德语键盘补丁程序；
15 *      Marc Corsini 添加了法文键盘补丁程序。
16 */
17
18 #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。
19
20 .text
21 .globl _keyboard_interrupt
22
23 /*
24 * these are for the keyboard read functions
25 */
26 /*
27 * 以下这些用于读键盘操作。
28 */
29 // size 是键盘缓冲区（缓冲队列）长度（字节数）。
30 size = 1024 // must be a power of two ! And MUST be the same
31 as in tty_io.c !!!! */
32 // 值必须是 2 的次方！并且与 tty_io.c 中的值匹配!!!! */
33
34 // 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）。
35 head = 4 // 缓冲区头指针字段在 tty_queue 结构中的偏移。
36 tail = 8 // 缓冲区尾指针字段偏移。
37 proc_list = 12 // 等待该缓冲队列的进程字段偏移。
38 buf = 16 // 缓冲区字段偏移。
39
40 // 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；
41 // 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。
42 // 每个字节标志中各位的含义见如下说明：
43 // (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。
28 mode: .byte 0 /* caps, alt, ctrl and shift mode */
29 leds: .byte 2 /* num-lock, caps, scroll-lock mode (nom-lock on) */
30 e0: .byte 0
31
32 /*
33 * con_int is the real interrupt routine that reads the
34 * keyboard scan-code and converts it into the appropriate
35 * ascii character(s).
36 */
/*
* 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。
37 _keyboard_interrupt:
38     pushl %eax
39     pushl %ebx
40     pushl %ecx
41     pushl %edx
42     push %ds
43     push %es
44     movl $0x10,%eax // 将 ds、es 段寄存器置为内核数据段。
45     mov %ax,%ds

```



```

46      mov %ax,%es
47      xorl %al,%al          /* %eax is scan code */ /* eax 中是扫描码 */
48      inb $0x60,%al        // 读取扫描码→al。
49      cmpb $0xe0,%al       // 扫描码是 0xe0 吗？若是则跳转到设置 e0 标志代码处。
50      je set_e0
51      cmpb $0xe1,%al       // 扫描码是 0xe1 吗？若是则跳转到设置 e1 标志代码处。
52      je set_e1
53      call key_table(,%eax,4) // 调用键处理程序 key_table + eax * 4 (参见 502 行)。
54      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。
55 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。
56      jmp lf               // 延迟一会。
57 1:      jmp lf
58 1:      orb $0x80,%al      // al 位 7 置位 (禁止键盘工作)。
59      jmp lf               // 再延迟一会。
60 1:      jmp lf
61 1:      outb %al,$0x61     // 使 PPI PB7 位置位。
62      jmp lf               // 延迟一会。
63 1:      jmp lf
64 1:      andb $0x7F,%al     // al 位 7 复位。
65      outb %al,$0x61       // 使 PPI PB7 位复位 (允许键盘工作)。
66      movb $0x20,%al       // 向 8259 中断芯片发送 EOI (中断结束) 信号。
67      outb %al,$0x20
68      pushl $0              // 控制台 tty 号=0，作为参数入栈。
69      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。
70      addl $4,%esp          // 丢弃入栈的参数，弹出保留的寄存器，并中断返回。
71      pop %es
72      pop %ds
73      popl %edx
74      popl %ecx
75      popl %ebx
76      popl %eax
77      iret
78 set_e0: movb $1,e0        // 收到扫描前导码 0xe0 时，设置 e0 标志 (位 0)。
79      jmp e0_e1
80 set_e1: movb $2,e0        // 收到扫描前导码 0xe1 时，设置 e1 标志 (位 1)。
81      jmp e0_e1
82
83 /*
84 * This routine fills the buffer with max 8 bytes, taken from
85 * %ebx:%eax. (%edx is high). The bytes are written in the
86 * order %al,%ah,%eal,%eah,%bl,%bh ... until %eax is zero.
87 */
/*
* 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，如果有就唤醒之。
88 put_queue:
89     pushl %ecx
90     pushl %edx                // 下句取控制台 tty 结构中读缓冲队列指针。
91     movl _table_list,%edx     # read-queue for console
92     movl head(%edx),%ecx      // 取队列头指针→ecx。
93 1:   movb %al,buf(%edx,%ecx)  // 将 al 中的字符放入头指针位置处。
94     incl %ecx                // 头指针前移 1 字节。
95     andl $size-1,%ecx        // 调整头指针。若超出缓冲区末端则绕回开始处。
96     cmpl tail(%edx),%ecx     # buffer full - discard everything
                                   // 头指针==尾指针吗？（即缓冲队列满了吗？）
97     je 3f                   // 如果已满，则后面未放入的字符全抛弃。
98     shrdl $8,%ebx,%eax       // 将 ebx 中 8 个比特右移 8 位到 eax 中，ebx 不变。
99     je 2f                   // 还有字符吗？若没有（等于 0）则跳转。
100    shrll $8,%ebx            // 将 ebx 值右移 8 位，并跳转到标号 1 继续操作。
101    jmp 1b
102 2:   movl %ecx,head(%edx)    // 若已将所有字符都放入队列，则保存头指针。
103     movl proc_list(%edx),%ecx // 该队列的等待进程指针？
104     testl %ecx,%ecx         // 检测是否有等待该队列的进程。
105     je 3f                   // 无，则跳转；
106     movl $0,(%ecx)         // 有，则唤醒进程（置该进程为就绪状态）。
107 3:   popl %edx
108     popl %ecx
109     ret
110
// 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 502 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。
111 ctrl: movb $0x04,%al        // 0x4 是 mode 中左 ctrl 键对应的比特位（位 2）。
112     jmp 1f
113 alt:  movb $0x10,%al        // 0x10 是 mode 中左 alt 键对应的比特位（位 4）。
114 1:   cmpb $0,e0             // e0 置位了吗（按下的是右边的 ctrl/alt 键吗）？
115     je 2f                   // 不是则转。
116     addb %al,%al            // 是，则改成置相应右键标志位（位 3 或位 5）。
117 2:   orb %al,mode           // 设置 mode 标志中对应的比特位。
118     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。
119 unctrl: movb $0x04,%al      // mode 中左 ctrl 键对应的比特位（位 2）。
120     jmp 1f
121 unalt: movb $0x10,%al      // 0x10 是 mode 中左 alt 键对应的比特位（位 4）。
122 1:   cmpb $0,e0             // e0 置位了吗（释放的是右边的 ctrl/alt 键吗）？
123     je 2f                   // 不是，则转。
124     addb %al,%al            // 是，则改成复位相应右键的标志位（位 3 或位 5）。
125 2:   notb %al               // 复位 mode 标志中对应的比特位。
126     andb %al,mode
127     ret
128

```

```

// 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。
129 lshift:
130     orb $0x01,mode           // 是左 shift 键按下，设置 mode 中位 0。
131     ret
132 unlshift:
133     andb $0xfe,mode          // 是左 shift 键松开，复位 mode 中位 0。
134     ret
135 rshift:
136     orb $0x02,mode           // 是右 shift 键按下，设置 mode 中位 1。
137     ret
138 unrshift:
139     andb $0xfd,mode          // 是右 shift 键松开，复位 mode 中位 1。
140     ret
141
// 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）。
142 caps:  testb $0x80,mode       // 测试 mode 中位 7 是否已置位（即在按下状态）。
143         jne 1f                // 如果已处于按下状态，则返回（186 行）。
144         xorb $4,leds           // 翻转 leds 标志中 caps-lock 比特位（位 2）。
145         xorb $0x40,mode        // 翻转 mode 标志中 caps 键按下的比特位（位 6）。
146         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。
147 set_leds:
148     call kb_wait              // 等待键盘控制器输入缓冲空。
149     movb $0xed,%al            /* set leds command */ /* 设置 LED 的命令 */
150     outb %al,$0x60             // 发送键盘命令 0xed 到 0x60 端口。
151     call kb_wait              // 等待键盘控制器输入缓冲空。
152     movb leds,%al             // 取 leds 标志，作为参数。
153     outb %al,$0x60             // 发送该参数。
154     ret
155 uncaps: andb $0x7f,mode        // caps 键松开，则复位 mode 中的对应位（位 7）。
156     ret
157 scroll:
158     xorb $1,leds              // scroll 键按下，则翻转 leds 中对应位（位 0）。
159     jmp set_leds              // 根据 leds 标志重新开启或关闭 LED 指示器。
160 num:   xorb $2,leds           // num 键按下，则翻转 leds 中的对应位（位 1）。
161     jmp set_leds              // 根据 leds 标志重新开启或关闭 LED 指示器。
162
163 /*
164  * curosr-key/numeric keypad cursor keys are handled here.
165  * checking for numeric keypad etc.
166  */
167 /*
168  * 这里处理方向键/数字小键盘方向键，检测数字小键盘等。
169  */
167 cursor:
168     subb $0x47,%al            // 扫描码是数字键盘上的键（其扫描码>=0x47）发出的？
169     jb 1f                     // 如果小于则不处理，返回（186 行）。
170     cmpb $12,%al              // 如果扫描码 > 0x53 (0x53 - 0x47 = 12)，则
171     ja 1f                     // 表示扫描码值超过 83 (0x53)，不处理，返回。
172     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。

```

```

173     testb $0x0c,mode      // 有 ctrl 键按下吗?
174     je cur2              // 无, 则跳转。
175     testb $0x30,mode      // 有 alt 键按下吗?
176     jne reboot          // 有, 则跳转到重启处理 (583 行)。
177 cur2: cmpb $0x01,e0      /* e0 forces cursor movement */ /* e0 置位光标移动*/
                                // e0 标志置位了吗?
178     je cur              // 置位了, 则跳转光标移动处理处 cur。
179     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。
180     je cur              // 若没有置位 (num 的 LED 不亮), 则也处理光标移动。
181     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。
182     jne cur             // 如果有 shift 键按下, 则也进行光标移动处理。
183     xorl %ebx,%ebx       // 否则查询小数字表 (199 行), 取键的数字 ASCII 码。
184     movb num_table(%eax),%al // 以 eax 作为索引值, 取对应数字字符→al。
185     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
186 1:     ret              // 在执行 put_queue 前需把 ebx 清零, 见 87 行上的注释。
187
    // 这段代码处理光标移动或插入删除按键。
188 cur:     movb cur_table(%eax),%al // 取光标字符表中相应键的代表字符→al。
189     cmpb $'9',%al      // 若字符≤'9' (5、6、2 或 3), 说明是上一页、下一页、
190     ja ok_cur          // 插入或删除键, 则功能字符序列中要添入字符'~'。不过
191     movb $'~',%ah      // 本内核并没有对这些键值进行识别和处理。
192 ok_cur: shll $16,%eax   // 将 ax 中内容移到 eax 高字中。
193     movw $0x5b1b,%ax   // 把'esc ['放入 ax, 与 eax 高字中字符组成移动序列。
194     xorl %ebx,%ebx     // 由于只需把 eax 中字符放入队列, 因此需要把 ebx 清零。
195     jmp put_queue     // 将该字符放入缓冲队列中。
196
197 #if defined(KBD_FR)
198 num_table:
199     .ascii "789 456 1230." // 数字小键盘上键对应的数字 ASCII 码表。
200 #else
201 num_table:
202     .ascii "789 456 1230,"
203 #endif
204 cur_table:
205     .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。
206
207 /*
208  * this routine handles function keys
209  */
210 /*
211  * 下面子程序处理功能键。
212  */
213 // 把功能键扫描码变换成转义字符序列并存放到读队列中。
214 func:
215     pushl %eax
216     pushl %ecx
217     pushl %edx
218     call _show_stat      // 调用显示各任务状态函数 (kern1/sched.c, 37 行)。
219     popl %edx
220     popl %ecx
221     popl %eax

```

```

218     subb $0x3B,%al          // 键'F1'的扫描码是 0x3B, 因此 al 中是功能键索引号。
219     jb end_func             // 如果扫描码小于 0x3b, 则不处理, 返回。
220     cmpb $9,%al            // 功能键是 F1-F10?
221     jbe ok_func             // 是, 则跳转。
222     subb $18,%al           // 是功能键 F11, F12 吗? F11、F12 扫描码是 0x57、0x58。
223     cmpb $10,%al           // 是功能键 F11?
224     jb end_func            // 不是, 则不处理, 返回。
225     cmpb $11,%al           // 是功能键 F12?
226     ja end_func            // 不是, 则不处理, 返回。
227 ok_func:
228     cmpl $4,%ecx            /* check that there is enough room */ /*检查空间*/
229     jl end_func             // [??]需要放入 4 个字符序列, 如果放不下, 则返回。
230     movl func_table(,%eax,4),%eax // 取功能键对应字符序列。
231     xorl %ebx,%ebx          // 因要放入队列字符数=4, 因此执行 put_queue 之前
232     jmp put_queue           // 需把 ebx 清零。
233 end_func:
234     ret
235
236 /*
237 * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
238 */
239 /*
240 * 功能键发送的扫描码, F1 键为: 'esc [ [ A', F2 键为: 'esc [ [ B' 等。
241 */
242 func_table:
243     .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
244     .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
245     .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
246
247 // 扫描码-ASCII 字符映射表。
248 // 根据 include/linux/config.h 文件中定义的键盘类型 (FINNISH, US, GERMEN, FRANCH), 将
249 // 相应键的扫描码映射到 ASCII 字符。
250 #if defined(KBD_FINNISH)
251 // 以下是芬兰语键盘的扫描码映射表。
252 key_map:
253     .byte 0,27              // 扫描码 0x00,0x01 对应的 ASCII 码;
254     .ascii "1234567890+' " // 扫描码 0x02,...0x0c,0x0d 对应的 ASCII 码, 以下类似。
255     .byte 127,9
256     .ascii "qwertyuiop}"
257     .byte 0,13,0
258     .ascii "asdfghjkl|{"
259     .byte 0,0
260     .ascii "'zxcvbnm,.-"
261     .byte 0,'*',0,32        /* 36-39 */ /* 扫描码 0x36-0x39 对应的 ASCII 码 */
262     .fill 16,1,0            /* 3A-49 */ /* 扫描码 0x3A-0x49 对应的 ASCII 码 */
263     .byte '-',0,0,0,'+'     /* 4A-4E */ /* 扫描码 0x4A-0x4E 对应的 ASCII 码 */
264     .byte 0,0,0,0,0,0,0     /* 4F-55 */ /* 扫描码 0x4F-0x55 对应的 ASCII 码 */
265     .byte '<'
266     .fill 10,1,0
267
268 shift_map:                  // shift 键同时按下时的映射表。
269     .byte 0,27
270     .ascii "!\"#$%&/'()=?`"

```

```

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

```

```

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

```



```

370     .byte 0, ''
371     .ascii "YXCVBNM;:_\"
372     .byte 0, '*', 0, 32          /* 36-39 */
373     .fill 16, 1, 0              /* 3A-49 */
374     .byte '-', 0, 0, 0, '+      /* 4A-4E */
375     .byte 0, 0, 0, 0, 0, 0, 0   /* 4F-55 */
376     .byte '>'
377     .fill 10, 1, 0
378
379 alt_map:
380     .byte 0, 0
381     .ascii "\0@\0$\0\0{[]}\0\"
382     .byte 0, 0
383     .byte '@, 0, 0, 0, 0, 0, 0, 0, 0, 0
384     .byte '~', 13, 0
385     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
386     .byte 0, 0
387     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
388     .byte 0, 0, 0, 0          /* 36-39 */
389     .fill 16, 1, 0          /* 3A-49 */
390     .byte 0, 0, 0, 0, 0      /* 4A-4E */
391     .byte 0, 0, 0, 0, 0, 0, 0 /* 4F-55 */
392     .byte '|'
393     .fill 10, 1, 0
394
395
396 #elif defined(KBD_FR)          // 以下是法语键盘的扫描码映射表。
397
398 key_map:
399     .byte 0, 27
400     .ascii "&{\\" (-) _/@)=\""
401     .byte 127, 9
402     .ascii "azertyuiop^$"
403     .byte 13, 0
404     .ascii "qsd fghjklm|"
405     .byte '`', 0, 42          /* coin sup gauche, don't know, [*|mu] */
406     .ascii "wxcvbn,;:!"
407     .byte 0, '*', 0, 32      /* 36-39 */
408     .fill 16, 1, 0          /* 3A-49 */
409     .byte '-', 0, 0, 0, '+   /* 4A-4E */
410     .byte 0, 0, 0, 0, 0, 0, 0 /* 4F-55 */
411     .byte '<'
412     .fill 10, 1, 0
413
414 shift_map:
415     .byte 0, 27
416     .ascii "1234567890]+\"
417     .byte 127, 9
418     .ascii "AZERTYUIOP<>\"
419     .byte 13, 0
420     .ascii "QSDFGHJKLM%\"
421     .byte '~', 0, '#'
422     .ascii "WXCVCBN?. /\\\"

```

```

423     .byte 0,'*',0,32          /* 36-39 */
424     .fill 16,1,0             /* 3A-49 */
425     .byte '-',0,0,0,'+'      /* 4A-4E */
426     .byte 0,0,0,0,0,0,0     /* 4F-55 */
427     .byte '>'
428     .fill 10,1,0
429
430 alt_map:
431     .byte 0,0
432     .ascii "\0~#{[|\`\\^@]}"
433     .byte 0,0
434     .byte '@,0,0,0,0,0,0,0,0,0
435     .byte '~',13,0
436     .byte 0,0,0,0,0,0,0,0,0,0
437     .byte 0,0
438     .byte 0,0,0,0,0,0,0,0,0,0
439     .byte 0,0,0,0           /* 36-39 */
440     .fill 16,1,0           /* 3A-49 */
441     .byte 0,0,0,0,0         /* 4A-4E */
442     .byte 0,0,0,0,0,0,0     /* 4F-55 */
443     .byte '|'
444     .fill 10,1,0
445
446 #else
447 #error "KBD-type not defined"
448 #endif
449 /*
450  * do_self handles "normal" keys, ie keys that don't change meaning
451  * and which have just one character returns.
452  */
453 /*
454  * 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。
455  */
456 do_self:
457     // 首先根据 mode 标志选择 alt_map、shift_map 或 key_map 映射表之一。
458     lea alt_map,%ebx          // 取 alt 键同时按下时的映射表基址 alt_map。
459     testb $0x20,mode          /* alt-gr */ /* 右 alt 键同时按下了? */
460     jne 1f                    // 是，则向前跳转到标号 1 处。
461     lea shift_map,%ebx        // 取 shift 键同时按下时的映射表基址 shift_map。
462     testb $0x03,mode          // 有 shift 键同时按下了吗?
463     jne 1f                    // 有，则向前跳转到标号 1 处。
464     lea key_map,%ebx          // 否则使用普通映射表 key_map。
465     // 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）。
466 1:     movb (%ebx,%eax),%al    // 将扫描码作为索引值，取对应的 ASCII 码→al。
467     orb %al,%al               // 检测看是否有对应的 ASCII 码。
468     je none                   // 若没有(对应的 ASCII 码=0)，则返回。
469     // 若 ctrl 键已按下或 caps 键锁定，并且字符在 'a'-'z' (0x61—0x7D) 范围内，则将其转成
470     // 大写字符 (0x41—0x5D)。
471     testb $0x4c,mode          /* ctrl or caps */ /* 控制键已按下或 caps 亮? */
472     je 2f                     // 没有，则向前跳转标号 2 处。
473     cmpb $'a',%al             // 将 al 中的字符与 'a' 比较。
474     jb 2f                     // 若 al 值 < 'a'，则转标号 2 处。
475     cmpb $'}',%al             // 将 al 中的字符与 '}' 比较。

```

```

469     ja 2f                                // 若 al 值>'', 则转标号 2 处。
470     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)。
471 2:     testb $0x0c,mode                 /* ctrl */ /* ctrl 键同时按下了吗? */
472     je 3f                                // 若没有则转标号 3。
473     cmpb $64,%al                        // 将 al 与 '@' (64) 字符比较, 即判断字符所属范围。
474     jb 3f                                // 若值<'@', 则转标号 3。
475     cmpb $64+32,%al                    // 将 al 与 ``' (96) 字符比较, 即判断字符所属范围。
476     jae 3f                              // 若值>='``, 则转标号 3。
477     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。
478 3:     testb $0x10,mode                 /* left alt */ /* 左 alt 键同时按下? */
479     je 4f                                // 没有, 则转标号 4。
480     orb $0x80,%al                      // 字符的位 7 置位。
// 将 al 中的字符放入读缓冲队列中。
481 4:     andl $0xff,%eax                  // 清 eax 的高字和 ah。
482     xorl %ebx,%ebx                      // 由于放入队列字符数<=4, 因此需把 ebx 清零。
483     call put_queue                      // 将字符放入缓冲队列中。
484 none:  ret
485
486 /*
487 * minus has a routine of it's own, as a 'E0h' before
488 * the scan code for minus means that the numeric keypad
489 * slash was pushed.
490 */
/*
* 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 '-' 字符键, 而美式键盘上是 '/' 键。
// 参见 L253、L355 和 L303 行。
491 minus: cmpb $1,e0                      // e0 标志置位了吗?
492     jne do_self                          // 没有, 则调用 do_self 对减号符进行普通处理。
493     movl $'/',%eax                      // 否则用 '/' 替换减号 '-' → al。
494     xorl %ebx,%ebx                      // 由于放入队列字符数<=4, 因此需把 ebx 清零。
495     jmp put_queue                       // 并将字符放入缓冲队列中。
496
497 /*
498 * This table decides which routine to call when a scan-code has been
499 * gotten. Most routines just call do_self, or none, depending if
500 * they are make or break.
501 */
/* 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, 这取决于按键
* (make) 还是释放键(break)。
*/
502 key_table:
503     .long none,do_self,do_self,do_self  /* 00-03 s0 esc 1 2 */
504     .long do_self,do_self,do_self,do_self /* 04-07 3 4 5 6 */
505     .long do_self,do_self,do_self,do_self /* 08-0B 7 8 9 0 */
506     .long do_self,do_self,do_self,do_self /* 0C-0F + ' bs tab */
507     .long do_self,do_self,do_self,do_self /* 10-13 q w e r */

```

```

508 . long do_self, do_self, do_self, do_self /* 14-17 t y u i */
509 . long do_self, do_self, do_self, do_self /* 18-1B o p } ^ */
510 . long do_self, ctrl, do_self, do_self /* 1C-1F enter ctrl a s */
511 . long do_self, do_self, do_self, do_self /* 20-23 d f g h */
512 . long do_self, do_self, do_self, do_self /* 24-27 j k l | */
513 . long do_self, do_self, lshift, do_self /* 28-2B { para lshift , */
514 . long do_self, do_self, do_self, do_self /* 2C-2F z x c v */
515 . long do_self, do_self, do_self, do_self /* 30-33 b n m , */
516 . long do_self, minus, rshift, do_self /* 34-37 . - rshift * */
517 . long alt, do_self, caps, func /* 38-3B alt sp caps f1 */
518 . long func, func, func, func /* 3C-3F f2 f3 f4 f5 */
519 . long func, func, func, func /* 40-43 f6 f7 f8 f9 */
520 . long func, num, scroll, cursor /* 44-47 f10 num scr home */
521 . long cursor, cursor, do_self, cursor /* 48-4B up pgup - left */
522 . long cursor, cursor, do_self, cursor /* 4C-4F n5 right + end */
523 . long cursor, cursor, cursor, cursor /* 50-53 dn pgdn ins del */
524 . long none, none, do_self, func /* 54-57 sysreq ? < f11 */
525 . long func, none, none, none /* 58-5B f12 ? ? ? */
526 . long none, none, none, none /* 5C-5F ? ? ? ? */
527 . long none, none, none, none /* 60-63 ? ? ? ? */
528 . long none, none, none, none /* 64-67 ? ? ? ? */
529 . long none, none, none, none /* 68-6B ? ? ? ? */
530 . long none, none, none, none /* 6C-6F ? ? ? ? */
531 . long none, none, none, none /* 70-73 ? ? ? ? */
532 . long none, none, none, none /* 74-77 ? ? ? ? */
533 . long none, none, none, none /* 78-7B ? ? ? ? */
534 . long none, none, none, none /* 7C-7F ? ? ? ? */
535 . long none, none, none, none /* 80-83 ? br br br */
536 . long none, none, none, none /* 84-87 br br br br */
537 . long none, none, none, none /* 88-8B br br br br */
538 . long none, none, none, none /* 8C-8F br br br br */
539 . long none, none, none, none /* 90-93 br br br br */
540 . long none, none, none, none /* 94-97 br br br br */
541 . long none, none, none, none /* 98-9B br br br br */
542 . long none, unctrl, none, none /* 9C-9F br unctrl br br */
543 . long none, none, none, none /* A0-A3 br br br br */
544 . long none, none, none, none /* A4-A7 br br br br */
545 . long none, none, unlshift, none /* A8-AB br br unlshift br */
546 . long none, none, none, none /* AC-AF br br br br */
547 . long none, none, none, none /* B0-B3 br br br br */
548 . long none, none, unrshift, none /* B4-B7 br br unrshift br */
549 . long unalt, none, uncaps, none /* B8-BB unalt br uncaps br */
550 . long none, none, none, none /* BC-BF br br br br */
551 . long none, none, none, none /* C0-C3 br br br br */
552 . long none, none, none, none /* C4-C7 br br br br */
553 . long none, none, none, none /* C8-CB br br br br */
554 . long none, none, none, none /* CC-CF br br br br */
555 . long none, none, none, none /* D0-D3 br br br br */
556 . long none, none, none, none /* D4-D7 br br br br */
557 . long none, none, none, none /* D8-DB br ? ? ? */
558 . long none, none, none, none /* DC-DF ? ? ? ? */
559 . long none, none, none, none /* E0-E3 e0 e1 ? ? */
560 . long none, none, none, none /* E4-E7 ? ? ? ? */

```

```

561     .long none, none, none, none          /* E8-EB ? ? ? ? */
562     .long none, none, none, none          /* EC-EF ? ? ? ? */
563     .long none, none, none, none          /* F0-F3 ? ? ? ? */
564     .long none, none, none, none          /* F4-F7 ? ? ? ? */
565     .long none, none, none, none          /* F8-FB ? ? ? ? */
566     .long none, none, none, none          /* FC-FF ? ? ? ? */
567
568 /*
569  * kb_wait waits for the keyboard controller buffer to empty.
570  * there is no timeout - if the buffer doesn't empty, we hang.
571  */
572 /*
573  * 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
574  * 缓冲永远不空的话，程序就会永远等待（死机）。
575  */
576 kb_wait:
577     pushl %eax
578     1:   inb $0x64,%al                      // 读键盘控制器状态。
579     testb $0x02,%al                      // 测试输入缓冲器是否为空（等于 0）。
580     jne 1b                               // 若不空，则跳转循环等待。
581     popl %eax
582     ret
583
584 /*
585  * This routine reboots the machine by asking the keyboard
586  * controller to pulse the reset-line low.
587  */
588 /*
589  * 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
590  * 位重启（reboot）。
591  */
592 // 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。
593 // 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
594 // 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
595 // 值为 0，则执行冷启动（Cold-boot）过程。
596 reboot:
597     call kb_wait                          // 首先等待键盘控制器输入缓冲器空。
598     movw $0x1234, 0x472                    /* don't do memory check */ /* 不进行内存检测 */
599     movb $0xfc,%al                        /* pulse reset and A20 low */
600     outb %al,$0x64                        // 向系统复位和 A20 线输出负脉冲。
601 die:   jmp die                            // 死机。

```

10.3.3 其他信息

10.3.3.1 PC/AT 键盘接口编程

PC 主板上的键盘接口是专用接口，它可以看作是常规串行端口的一个简化版本。该接口被称为键盘控制器，它使用串行通信协议接收键盘发来的扫描码数据。主板上所采用的键盘控制器是 Intel 8042 芯片或其兼容芯片，其逻辑示意图见图 10-7 所示。现今的主板上已经不包括独立的 8042 芯片了，但是主板上其他集成电路会为兼容目的而模拟 8042 芯片的功能。另外，该芯片输出端口 P2 各位被分别用于其他目的。位 0（P20 引脚）用于实现 CPU 的复位操作，位 1（P21 引脚）用于控制 A20 信号线的开启与否。当该输出端口位 1 为 1 时就开启（选通）了 A20 信号线，为 0 则禁止 A20 信号线。参见引导启动程序一章中对 A20 信号线的详细说明。

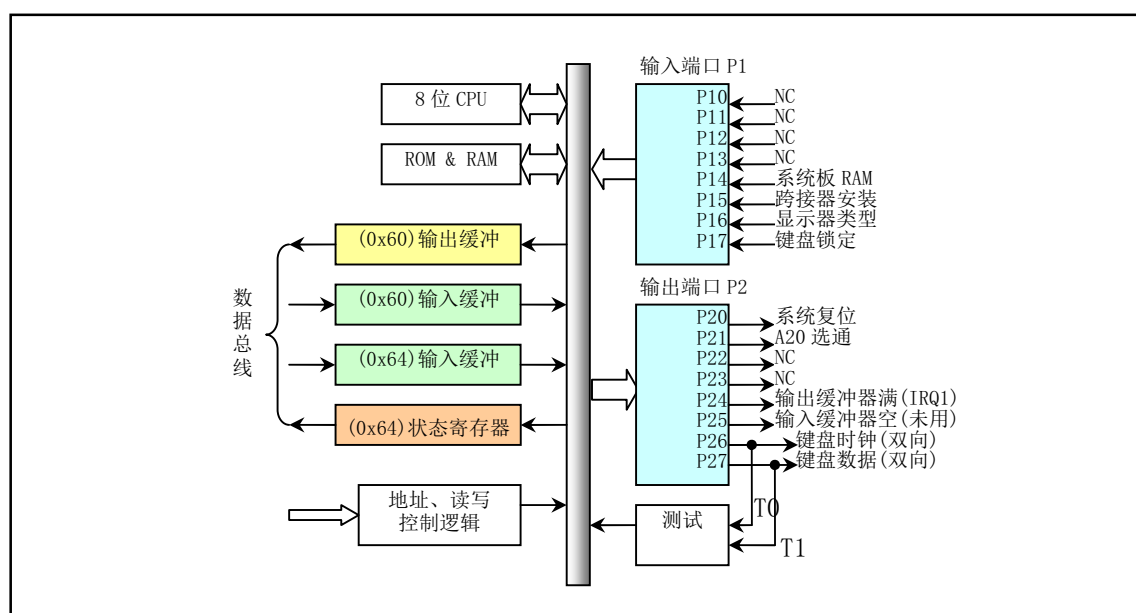


图 10-7 键盘控制器 804X 逻辑示意图

分配给键盘控制器的 IO 端口范围是 0x60-0x6f，但实际上 IBM CP/AT 使用的只有 0x60 和 0x64 两个口地址（0x61、0x62 和 0x63 用于与 XT 兼容目的）见表 10-1 所示，加上对端口的读和写操作含义不同，因此主要可有 4 种不同操作。对键盘控制器进行编程，将涉及芯片中的状态寄存器、输入缓冲器和输出缓冲器。

表 10-1 键盘控制器 804X 端口

端口	读/写	名称	用途
0x60	读	数据端口或输出缓冲器	是一个 8 位只读寄存器。当键盘控制器收到来自键盘的扫描码或命令响应时，一方面置状态寄存器位 0 = 1，另一方面产生中断 IRQ1。通常应该仅在状态端口位 0 = 1 时才读。
0x60	写	输入缓冲器	用于向键盘发送命令与/或随后的参数，或向键盘控制器写参数。键盘命令共有 10 多条，见表格后说明。通常都应该仅在状态端口位 1=0 时才写。
0x61	读/写		该端口 0x61 是 8255A 输出口 B 的地址，是针对使用/兼容 8255A 的 PC 标准键盘电路进行硬件复位处理。该端口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。所操作的数据为： 位 7=1 禁止键盘；=0 允许键盘； 位 6=0 迫使键盘时钟为低位，因此键盘不能发送任何数据。 位 5-0 这些位与键盘无关，是用于可编程并行接口(PPI)。
0x64	读	状态寄存器	是一个 8 位只读寄存器，其位字段含义分别为： 位 7=1 来自键盘传输数据奇偶校验错； 位 6=1 接收超时(键盘传送未产生 IRQ1)； 位 5=1 发送超时(键盘无响应)； 位 4=1 键盘接口被键盘锁禁止；[??是=0 时] 位 3=1 写入输入缓冲器中的数据是命令(通过端口 0x64)； =0 写入输入缓冲器中的数据是参数(通过端口 0x60)； 位 2 系统标志状态：0 = 上电启动或复位；1 = 自检通过；

			位 1=1 输入缓冲器满(0x60/64 口有给 8042 的数据); 位 0=1 输出缓冲器满(数据端口 0x60 有给系统的数据)。
0x64	写	输入缓冲器	向键盘控制器写命令。可带一参数, 参数从端口 0x60 写入。键盘控制器命令有 12 条, 见表格后说明。

10.3.3.2 键盘命令

系统在向端口 0x60 写入 1 字节, 便是发送键盘命令。键盘在接收到命令后 20ms 内应予以响应, 即回送一个命令响应。有的命令后还需要跟一参数 (也写到该端口)。命令列表见表 10-2 所示。注意, 如果没有另外指明, 所有命令均被回送一个 0xfa 响应码(ACK)。

表 10-2 键盘命令一览表

命令码	参数	功能
0xed	有	设置/复位模式指示器。置 1 开启, 0 关闭。参数字节: 位 7-3 保留全为 0; 位 2 = caps-lock 键; 位 1 = num-lock 键; 位 0 = scroll-lock 键。
0xee	无	诊断回应。键盘应回送 0xee。
0xef		保留不用。
0xf0	有	读取/设置扫描码集。参数字节等于: 0x00 - 选择当前扫描码集; 0x01 - 选择扫描码集 1(用于 PCs, PS/2 30 等); 0x02 - 选择扫描码集 2(用于 AT, PS/2, 是缺省值); 0x03 - 选择扫描码集 3。
0xf1		保留不用。
0xf2	无	读取键盘标识号(读取 2 个字节)。AT 键盘返回响应码 0xfa。
0xf3	有	设置扫描码连续发送时的速率和延迟时间。参数字节的含义为: 位 7 保留为 0; 位 6-5 延时值: 令 $C = \text{位 } 6-5$, 则有公式: 延时值 $= (1+C) \times 250\text{ms}$; 位 4-0 扫描码连续发送的速率; 令 $B = \text{位 } 4-3$; $A = \text{位 } 2-0$, 则有公式: 速率 $= 1 / ((8+A) \times 2^B \times 0.00417)$ 。 参数缺省值为 0x2c。
0xf4	无	开启键盘。
0xf5	无	禁止键盘。
0xf6	无	设置键盘默认参数。
0xf7-0xfd		保留不用。
0xfe	无	重发扫描码。当系统检测到键盘传输数据有错, 则发此命令。
0xff	无	执行键盘上电复位操作, 称之为基本保证测试(BAT)。操作过程为: 1. 键盘收到该命令后立刻响应发送 0xfa; 2. 键盘控制器使键盘时钟和数据线置为高电平; 3. 键盘开始执行 BAT 操作; 4. 若正常完成, 则键盘发送 0xaa; 否则发送 0xfd 并停止扫描。

10.3.3.3 键盘控制器命令

系统向输入缓冲(端口 0x64)写入 1 字节,即发送一键盘控制器命令。可带一参数。参数是通过写 0x60 端口发送的。见表 10-3 所示。

表 10-3 键盘控制器命令一览表

命令	参数	功能
0x20	无	读给键盘控制器的最后一个命令字节, 放在端口 0x60 供系统读取。
0x21-0x3f	无	读取由命令低 5 比特位指定的控制器内部 RAM 中的命令。
0x60-0x7f	有	写键盘控制器命令字节。参数字节: (默认值为 0x5d) 位 7 保留为 0; 位 6 IBM PC 兼容模式(奇偶检验, 转换为系统扫描码, 单字节 PC 断开码); 位 5 PC 模式 (对扫描码不进行奇偶校验; 不转换成系统扫描码); 位 4 禁止键盘工作 (使键盘时钟为低电平); 位 3 禁止超越(override), 对键盘锁定转换不起作用; 位 2 系统标志; 1 表示控制器工作正确; 位 1 保留为 0; 位 0 允许输出寄存器满中断。
0xaa	无	初始化键盘控制器自测试。成功返回 0x55; 失败返回 0xfc。
0xab	无	初始化键盘接口测试。返回字节: 0x00 无错; 0x01 键盘时钟线为低(始终为低, 低粘连); 0x02 键盘时钟线为高; 0x03 键盘数据线为低; 0x04 键盘数据线为高;
0xac	无	诊断转储。804x 的 16 字节 RAM、输出口、输入口状态依次输出给系统。
0xad	无	禁止键盘工作 (设置命令字节位 4=1)。
0xae	无	允许键盘工作 (复位命令字节位 4=0)。
0xc0	无	读 804x 的输入端口 P1, 并放在 0x60 供读取;
0xd0	无	读 804x 的输出端口 P2, 并放在 0x60 供读取;
0xd1	有	写 804x 的输出端口 P2, 原 IBM PC 使用输出端口的位 2 控制 A20 门。注意, 位 0(系统复位)应该总是置位的。
0xe0	无	读测试端 T0 和 T1 的输入送输出缓冲器供系统读取。 位 1 键盘数据; 位 0 键盘时钟。
0xed	有	控制 LED 的状态。置 1 开启, 0 关闭。参数字节: 位 7-3 保留全为 0; 位 2 = caps-lock 键; 位 1 = num-lock 键; 位 0 = scroll-lock 键。
0xf0-0xff	无	送脉冲到输出端口。该命令序列控制输出端口 P20-23 线, 参见键盘控制器逻辑示意图。欲让哪一位输出负脉冲(6 微秒), 即置该位为 0。也即该命令的低 4 位分别控制负脉冲的输出。例如, 若要复位系统, 则需发出命令 0xfe(P20 低)即可。

10.3.3.4 键盘扫描码

PC 机采用的均是非编码键盘。键盘上每个键都有一个位置编号, 是从左到右从上到下。并且 PC XT

机与 AT 机键盘的位置码差别很大。键盘内的微处理机向系统发送的是键对应的扫描码。当键按下时，键盘输出的扫描码称为接通(make)扫描码，而该键松开时发送的则称为断开(break)扫描码。XT 键盘各键的扫描码见表 10-4 所示。

表 10-4 XT 键盘扫描码表

F1	F2															BS	ESC	NUML	SCRL	SYSR
3B	3C	29	02	03	04	05	06	07	08	09	0A	0B	0C	0D	2B	0E	01	45	46	**
F3	F4	TAB	Q	W	E	R	T	Y	U	I	O	P	[]			Home	↑	PgUp	PrtSc
3D	3E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B			47	48	49	37
F5	F6	CNTL	A	S	D	F	G	H	J	K	L	;	'	ENTER		←	5	→	-	
3F	40	1D	1E	1F	20	21	22	23	24	25	26	27	28	1C		4B	4C	4D	4A	
F7	F8	LSHFT		Z	X	C	V	B	N	M	,	.	/	RSHFT		End	↓	PgDn	+	
41	42	2A		2C	2D	2E	2F	30	31	32	33	34	35	36		4F	50	51	4E	
F9	F10	ALT			Space											CAPLOCK	Ins		Del	
43	44	38			39											3A	52		53	

键盘上的每个键都有一个包含在字节低 7 位（位 6-0）中相应的扫描码。在高位（位 7）表示是按键还是松开按键。位 7=0 表示刚将键按下的扫描码，位 7=1 表示键松开的扫描码。例如，如果某人刚把 ESC 键按下，则传输给系统的扫描码将是 1（1 是 ESC 键的扫描码），当该键释放时将产生 $1+0x80=129$ 扫描码。

对于 PC、PC/XT 的标准 83 键键盘，接通扫描码与键号（键的位置码）是一样的。并用 1 字节表示。例如“A”键，键位置号是 30，接通码和扫描码也是 30（0x1e）。而其断开码是接通扫描码加上 0x80，即 0x9e。对于 AT 机使用的 84/101/102 扩展键盘，则与 PC/XT 标准键盘区别较大。

对于某些“扩展的”键则情况有些不同。当一个扩展键被按下时，将产生一个中断并且键盘端口将输出一个“扩展的”的扫描码前缀 0xe0，而在下一个中断中则会给出“扩展的”扫描码。比如，对于 PC/XT 标准键盘，左边的控制键 ctrl 的扫描码是 29（0x1d），而右边的“扩展的”控制键 ctrl 则具有一个扩展的扫描码序列 0xe0、0x1d。这个规则同样适合于 alt、箭头键。

另外，还有两个键的处理非常特殊，PrtScn 键和 Pause/Break 键。按下 PrtScn 键将会向键盘中断程序发送 2 个扩展字符，42（0x2a）和 55（0x37），所以实际的字节序列将是 0xe0，0x2a，0xe0，0x37。但在键重复产生时还会发送扩展字符 0xaa，即产生序列 0xe0，0x2a，0xe0，0x37，0xe0，0xaa。当键松开时，又重新发送两个扩展的加上 0x80 的码（0xe0，0xb7，0xe0，0xaa）。当 prtscn 键按下时，如果 shift 或 ctrl 键也按下了，则仅发送 0xe0，0x37，并且在松开时仅发送 0xe0，0xb7。如果按下了 alt 键，那么 PrtScn 键就如同一个具有扫描码 0x54 的普通键。

对于 Pause/Break 键。如果你在按下该键的同时也按下了任意一个控制键 ctrl，则将行如扩展键 70（0x46），而在其他情况下它将发送字符序列 0xe1，0x1d，0x45，0xe1，0x9d，0xc5。将键一直按下并不会产生重复的扫描码，而松开键也并不会产生任何扫描码。因此，我们可以这样来看待和处理：扫描码 0xe0 意味着还有一个字符跟随其后，而扫描码 0xe1 则表示后面跟随着 2 个字符。

对于 AT 键盘的扫描码，与 PC/XT 的略有不同。当键按下时，则对应键的扫描码被送出，但当键松开时，将会发送两个字节，第一个是 0xf0，第 2 个还是相同的键扫描码。现在键盘设计者使用 8049 作为 AT 键盘的输入处理器，为了向下的兼容性将 AT 键盘发出的扫描码转换成了老式 PC/XT 标准键盘的扫描码。

AT 键盘有三种独立的扫描码集：一种是我们上面说明的(83 键映射，而增加的键有多余的 0xe0 码)，一种几乎是顺序的，还有一种却只有 1 个字节！最后一种所带来的问题是只有左 shift，caps，左 ctrl 和左 alt 键的松开码被发送。键盘的默认扫描码集是扫描码集 2，可以利用命令更改。

对于扫描码集 1 和 2，有特殊码 0xe0 和 0xe1。它们用于具有相同功能的键。比如：左控制键 ctrl 位置是 0x1d(对于 PC/XT)，则右边的控制键就是 0xe0，0x1d。这是为了与 PC/XT 程序兼容。请注意唯一

使用 0xe1 的时候是当它表示临时控制键时，对此情况同时也有一个 0xe0 的版本。

10.4 console.c 程序

10.4.1 功能描述

本文件是内核中最长的程序之一，但功能比较单一。其中的所有子程序都是为了实现终端屏幕写函数 `con_write()` 以及进行终端屏幕显示的控制操作。

当往一个控制台设备执行写操作时，就会调用 `con_write()` 函数。这个函数管理所有控制字符和换码字符序列，这些字符给应用程序提供全部的屏幕管理操作。所实现的换码序列是 vt102 终端的；这意味着当你使用 telnet 连接到一台非 Linux 主机时，你的环境变量应该有 `TERM=vt102`；然而，对于本地操作最佳的选择是设置 `TERM=console`，因为 Linux 控制台提供了一个 vt102 功能的超集。

函数 `con_write()` 主要由转换语句组成，用于每次处理一个字符的有限长状态自动转义序列的解释。在正常方式下，显示字符使用当前属性直接写到显示内存中。该函数会从终端 `tty_struct` 结构的写缓冲队列 `write_q` 中取出字符或字符序列，然后根据字符的性质（是普通字符、控制字符、转义序列还是控制序列），把字符显示在终端屏幕上或进行一些光标移动、字符擦除等屏幕控制操作。

终端屏幕初始化函数 `con_init()` 会根据系统初始化时获得的系统信息，设置有关屏幕的一些基本参数值，用于 `con_write()` 函数的操作。

有关终端设备字符缓冲队列的说明可参见 `include/linux/tty.h` 头文件。其中给出了字符缓冲队列的数据结构 `tty_queue`、终端的数据结构 `tty_struct` 和一些控制字符的值。另外还有一些对缓冲队列进行操作的宏定义。缓冲队列及其操作示意图请参见图 10-14 所示。

10.4.2 代码注释

程序 10-3 linux/kernel/chr_drv/console.c

```

1  /*
2  *  linux/kernel/console.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *      console.c
9  *
10 *  This module implements the console io functions
11 *      'void con_init(void)'
12 *      'void con_write(struct tty_queue * queue)'
13 *  Hopefully this will be a rather complete VT102 implementation.
14 *
15 *  Beeping thanks to John T Kohl.
16 */
/*
 * 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。
    */
17
18 /*
19  * NOTE!!! We sometimes disable and enable interrupts for a short while
20  * (to put a word in video IO), but this will work even for keyboard
21  * interrupts. We know interrupts aren't enabled when getting a keyboard
22  * interrupt, as we use trap-gates. Hopefully all is well.
23  */
    /*
    * 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
    * 一个键盘中断过程中中断被禁止。希望一切均正常。
    */
24
25 /*
26  * Code to check for different video-cards mostly by Galen Hunt,
27  * <g-hunt@ee.utah.edu>
28  */
    /*
    * 检测不同显示卡的大多数代码是 Galen Hunt 编写的，
    * <g-hunt@ee.utah.edu>
    */
29
30 #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。
31 #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。
32 #include <asm/io.h> // io 头文件。定义硬件端口输入/输出宏汇编语句。
33 #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。
34
35 /*
36  * These are set up by the setup-routine at boot-time:
37  */
    /*
    * 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。
38
39 #define ORIG_X (* (unsigned char *) 0x90000) // 初始光标列号。
40 #define ORIG_Y (* (unsigned char *) 0x90001) // 初始光标行号。
41 #define ORIG_VIDEO_PAGE (* (unsigned short *) 0x90004) // 显示页面。
42 #define ORIG_VIDEO_MODE ((*(unsigned short *) 0x90006) & 0xff) // 显示模式。
43 #define ORIG_VIDEO_COLS (((*(unsigned short *) 0x90006) & 0xff00) >> 8) // 字符列数。
44 #define ORIG_VIDEO_LINES (25) // 显示行数。
45 #define ORIG_VIDEO_EGA_AX (* (unsigned short *) 0x90008) // [??]
46 #define ORIG_VIDEO_EGA_BX (* (unsigned short *) 0x9000a) // 显示内存大小和色彩模式。
47 #define ORIG_VIDEO_EGA_CX (* (unsigned short *) 0x9000c) // 显示卡特性参数。
48
    // 定义显示器单色/彩色显示模式类型符号常数。
49 #define VIDEO_TYPE_MDA 0x10 /* Monochrome Text Display */ /* 单色文本 */
50 #define VIDEO_TYPE_CGA 0x11 /* CGA Display */ /* CGA 显示器 */
51 #define VIDEO_TYPE_EGAM 0x20 /* EGA/VGA in Monochrome Mode */ /* EGA/VGA 单色 */
52 #define VIDEO_TYPE_EGAC 0x21 /* EGA/VGA in Color Mode */ /* EGA/VGA 彩色 */
53

```

```

54 #define NPAR 16                                // 转义字符序列中最大参数个数。
55
56 extern void keyboard_interrupt(void);           // 键盘中断处理程序 (keyboard.S)。
57
58 // 以下这些静态变量是本程序中使用的一些全局变量。
59 static unsigned char video_type;                /* Type of display being used */
60 /* 使用的显示类型 */
61 static unsigned long video_num_columns;         /* Number of text columns */
62 /* 屏幕文本列数 */
63 static unsigned long video_size_row;           /* Bytes per row */
64 /* 屏幕每行使用的字节数 */
65 static unsigned long video_num_lines;          /* Number of test lines */
66 /* 屏幕文本行数 */
67 static unsigned char video_page;               /* Initial video page */
68 /* 初始显示页面 */
69 static unsigned long video_mem_start;          /* Start of video RAM */
70 /* 显示内存起始地址 */
71 static unsigned long video_mem_end;            /* End of video RAM (sort of) */
72 /* 显示内存结束(末端)地址 */
73 static unsigned short video_port_reg;          /* Video register select port */
74 /* 显示控制索引寄存器端口 */
75 static unsigned short video_port_val;          /* Video register value port */
76 /* 显示控制数据寄存器端口 */
77 static unsigned short video_erase_char;        /* Char+Attrib to erase with */
78 /* 擦除字符属性及字符 (0x0720) */
79
80 // 以下这些变量用于屏幕滚屏操作。(origin 表示移动的虚拟窗口左上角原点内存地址)。
81 static unsigned long origin;                    /* Used for EGA/VGA fast scroll */
82 /* 用于 EGA/VGA 快速滚屏 */ // 滚屏起始内存地址。
83 static unsigned long scr_end;                  /* Used for EGA/VGA fast scroll */
84 /* 用于 EGA/VGA 快速滚屏 */ // 滚屏末端内存地址。
85 static unsigned long pos;                      // 当前光标对应的显示内存位置。
86 static unsigned long x, y;                    // 当前光标位置。
87 static unsigned long top, bottom;              // 滚动时顶行行号; 底行行号。
88 // state 用于标明处理 ESC 转义序列时的当前步骤。par[] 用于存放 ESC 序列的中间处理参数。
89 static unsigned long state=0;                  // ANSI 转义字符序列处理状态。
90 static unsigned long npar, par[NPAR];          // ANSI 转义字符序列参数个数和参数数组。
91 static unsigned long ques=0;                   // 收到问号字符标志。
92 static unsigned char attr=0x07;                // 字符属性 (黑底白字)。
93
94 static void sysbeep(void);                     // 系统蜂鸣函数。
95
96 /*
97  * this is what the terminal answers to a ESC-Z or csi0c
98  * query (= vt100 response).
99  */
100 // 下面是终端回应 ESC-Z 或 csi0c 请求的应答 (=vt100 响应)。
101 // csi - 控制序列引导码(Control Sequence Introducer)。
102 // 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' )
103 // 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
104 // 主机。该序列 (即 'ESC [?1;2c' ) 表示终端是高级视频终端。

```

```

85 #define RESPONSE "\033[?1;2c"
86
87 /* 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 */
    //// 跟踪光标当前位置。
    // 参数: 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。
88 static inline void gotoxy(unsigned int new_x, unsigned int new_y)
89 {
    // 首先检查参数的有效性。如果给定的光标列号超出显示器列数, 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应在显示内存中位置 pos。
90     if (new_x > video_num_columns || new_y >= video_num_lines)
91         return;
92     x=new_x;
93     y=new_y;
94     pos=origin + y*video_size_row + (x<<1); // 1 列用 2 个字节表示, 所以 x<<1。
95 }
96
    //// 设置滚屏起始显示内存地址。
97 static inline void set_origin(void)
98 {
    // 首先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, 然后
    // 写入滚屏起始地址高字节。向右移动 9 位, 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。输出值是相对于默认显示内存起始位置
    // video_mem_start 操作的, 例如对于彩色模式, viedo_mem_start = 物理内存地址 0xb8000。
99     cli();
100     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。
101     outb_p(0xff&((origin-video_mem_start)>>9), video_port_val);
102     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。
103     outb_p(0xff&((origin-video_mem_start)>>1), video_port_val);
104     sti();
105 }
106
    //// 向上卷动一行。
    // 将屏幕滚动区域中内容向下移动一行, 并在区域顶出现的新行上添加空格字符。滚屏区域必须
    // 起码是 2 行或 2 行以上。参见程序列表后说明。
107 static void scrup(void)
108 {
    // 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。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。
109     if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
110     {
        // 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。
        // 最后把新屏幕滚动窗口内存起始位置值 origin 写入显示控制器中。
111         if (!top && bottom == video_num_lines) {
112             origin += video_size_row;
113             pos += video_size_row;
114             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。
115         if (scr_end > video_mem_end) {
116             __asm__( "cld\n\t"           // 清方向位。
117                     "rep\n\t"           // 重复操作, 将当前屏幕内存
118                     "movsl\n\t"         // 数据移动到显示内存起始处。
119                     "movl _video_num_columns, %1\n\t"
120                     "rep\n\t"           // 在新行上填入空格字符。
121                     "stosw"
122                     :: "a" (video_erase_char),
123                     "c" ((video_num_lines-1)*video_num_columns>>1),
124                     "D" (video_mem_start),
125                     "S" (origin)
126                     : "cx", "di", "si");
127             scr_end -= origin-video_mem_start;
128             pos -= origin-video_mem_start;
129             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(屏幕字符行数); %2 -edi(最后 1 行开始处对应内存位置);
130         } else {
131             __asm__( "cld\n\t"
132                     "rep\n\t"           // 重复操作, 在新出现行上
133                     "stosw"           // 填入擦除字符(空格字符)。
134                     :: "a" (video_erase_char),
135                     "c" (video_num_columns),
136                     "D" (scr_end-video_size_row)
137                     : "cx", "di");
138         }
// 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。
139         set_origin();
// 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)。
140     } else {
141         __asm__( "cld\n\t"
142                 "rep\n\t"           // 循环操作, 将 top+1 到 bottom 行
143                 "movsl\n\t"         // 所对应的内存块移到 top 行开始处。
144                 "movl _video_num_columns, %%ecx\n\t"
145                 "rep\n\t"           // 在新行上填入擦除字符。
146                 "stosw"
147                 :: "a" (video_erase_char),
148                 "c" ((bottom-top-1)*video_num_columns>>1),
149                 "D" (origin+video_size_row*top),

```



```

150         "S" (origin+video_size_row*(top+1))
151         : "cx", "di", "si");
152     }
153 }

// 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。
154     else                /* Not EGA/VGA */
155     {
156         __asm__( "cld\n\t"
157                 "rep\n\t"
158                 "movsl\n\t"
159                 "movl _video_num_columns, %%ecx\n\t"
160                 "rep\n\t"
161                 "stosw"
162                 :: "a" (video_erase_char),
163                 "c" ((bottom-top-1)*video_num_columns>>1),
164                 "D" (origin+video_size_row*top),
165                 "S" (origin+video_size_row*(top+1))
166                 : "cx", "di", "si");
167     }
168 }
169

//// 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！
170 static void scrdown(void)
171 {
172     // 本函数也针对两类显示类型 (EGA/VGA 和 MDA) 分别进行操作。如果显示类型是 EGA，则执行
173     // 下列操作。(这里 if 和 else 中的操作完全一样！以后的内核就合二为一了。)
174     if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
175     {
176         // %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);
177         // %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)。
178         // 移动方向: [esi]→[edi], 移动 ecx 个长字。
179         __asm__( "std\n\t"           // 置方向位!!
180                 "rep\n\t"           // 重复操作，向下移动从 top 行到
181                 "movsl\n\t"         // bottom-1 行对应的内存数据。
182                 "addl $2, %%edi\n\t" /* %edi has been decremented by 4 */
183                                     /* %edi 已减 4，因也是反向填擦除字符*/
184                 "movl _video_num_columns, %%ecx\n\t"
185                 "rep\n\t"           // 将擦除字符填入上方新行中。
186                 "stosw"
187                 :: "a" (video_erase_char),
188                 "c" ((bottom-top-1)*video_num_columns>>1),
189                 "D" (origin+video_size_row*bottom-4),
190                 "S" (origin+video_size_row*(bottom-1)-4)
191                 : "ax", "cx", "di", "si");

```

```

186     }
187     // 如果不是 EGA 显示类型，则执行以下操作（与上面完全一样）。
188     else                /* Not EGA/VGA */
189     {
190         __asm__( "std\n\t"
191                 "rep\n\t"
192                 "movsl\n\t"
193                 "addl $2,%%edi\n\t" /* %edi has been decremented by 4 */
194                 "movl _video_num_columns, %%ecx\n\t"
195                 "rep\n\t"
196                 "stosw"
197                 "::" "a" (video_erase_char),
198                 "c" ((bottom-top-1)*video_num_columns>>1),
199                 "D" (origin+video_size_row*bottom-4),
200                 "S" (origin+video_size_row*(bottom-1)-4)
201                 : "ax", "cx", "di", "si");
202     }
203
204     // 光标位置下移一行 (lf - line feed 换行)。
205     // 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
206     // 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。
207     // 函数名称 lf (line feed 换行) 是指处理控制字符 LF。
208     static void lf(void)
209     {
210         if (y+1<bottom) {
211             y++;
212             pos += video_size_row;          // 加上屏幕一行占用内存的字节数。
213             return;
214         }
215         scrup();                            // 将屏幕窗口内容上移一行。
216     }
217
218     // 光标在同列上移一行。
219     // 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
220     // 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。
221     // 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"。
222     static void ri(void)
223     {
224         if (y>top) {
225             y--;
226             pos -= video_size_row;          // 减去屏幕一行占用内存的字节数。
227             return;
228         }
229         scrdwn();                          // 将屏幕窗口内容下移一行。
230     }
231
232     // 光标回到第 1 列 (0 列)。
233     // 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。
234     // 函数名称 cr (carriage return 回车) 指明处理的控制字符是回车字符。
235     static void cr(void)
236     {
237         pos -= x<<1;                      // 减去 0 列到光标处占用的内存字节数。

```

```

227     x=0;
228 }
229
230 // 擦除光标前一字符（用空格替代）（del - delete 删除）。
231 // 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），
232 // 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。
233 static void del(void)
234 {
235     if (x) {
236         pos -= 2;
237         x--;
238         *(unsigned short *)pos = video_erase_char;
239     }
240 }
241
242 // 删除屏幕上与光标位置相关的部分。
243 // 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；
244 // 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，
245 // 并且在擦除字符或行时光标位置不变。
246 // 函数名称 csi_J（CSI - Control Sequence Introducer，即控制序列引导码）指明对控制
247 // 序列“CSI Ps J”进行处理。
248 // 参数：par - 对应上面控制序列中 Ps 的值。
249 static void csi_J(int par)
250 {
251     long count __asm__("cx"); // 设为寄存器变量。
252     long start __asm__("di");
253
254     // 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。
255     switch (par) {
256     case 0: /* erase from cursor to end of display */
257         count = (scr_end-pos)>>1; /* 擦除光标到屏幕底端所有字符 */
258         start = pos;
259         break;
260     case 1: /* erase from start to cursor */
261         count = (pos-origin)>>1; /* 删除从屏幕开始到光标处的字符 */
262         start = origin;
263         break;
264     case 2: /* erase whole display */ /* 删除整个屏幕上的所有字符 */
265         count = video_num_columns * video_num_lines;
266         start = origin;
267         break;
268     default:
269         return;
270     }
271
272     // 然后使用擦除字符填写被删除字符的地方。
273     // %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)。
274     __asm__("cld\n\t"
275            "rep\n\t"
276            "stosw\n\t"
277            ":\n\t"
278            "c" count,
279            "D" (start), "a" (video_erase_char)
280            : "cx", "di");
281 }

```

```

267
268 // 删除一行上与光标位置相关的部分。
269 // 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)。
270 // 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
271 // 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。
272 // 参数: par - 对应上面控制序列中 Ps 的值。
273 static void csi_K(int par)
274 {
275     long count __asm__ ("cx");          // 设置寄存器变量。
276     long start __asm__ ("di");
277
278     // 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。
279     switch (par) {
280     case 0: /* erase from cursor to end of line */
281         if (x>=video_num_columns)      /* 删除光标到行尾所有字符 */
282             return;
283         count = video_num_columns - x;
284         start = pos;
285         break;
286     case 1: /* erase from start of line to cursor */
287         start = pos - (x<<1);          /* 删除从行开始到光标处 */
288         count = (x<video_num_columns)?x:video_num_columns;
289         break;
290     case 2: /* erase whole line */        /* 将整行字符全删除 */
291         start = pos - (x<<1);
292         count = video_num_columns;
293         break;
294     default:
295         return;
296     }
297
298     // 然后使用擦除字符填写删除字符的地方。
299     // %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)。
300     __asm__ ("cld\n\t"
301             "rep\n\t"
302             "stosw\n\t"
303             ":: \"c\" (count),
304             "D\" (start), \"a\" (video_erase_char)
305             : \"cx\", \"di\");
306
307     // 设置显示字符属性。
308     // ANSI 转义序列: 'ESC [ Ps m'。Ps =0 默认属性; 1 加粗; 4 加下划线; 7 反显; 27 正显。
309     // 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
310     // 性, 直到再次执行本控制序列重新设置字符显示的属性。对于单色和彩色显示卡, 设置的属
311     // 性是有区别的, 这里仅作了简化处理。
312     void csi_m(void)
313     {
314         int i;
315
316         for (i=0; i<=npar; i++)
317             switch (par[i]) {
318             case 0: attr=0x07; break;
319             case 1: attr=0x0f; break;

```

```

307             case 4:attr=0x0f;break;
308             case 7:attr=0x70;break;
309             case 27:attr=0x07;break;
310         }
311     }
312
313     // 设置显示光标。
314     // 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。
315     static inline void set_cursor(void)
316     {
317         // 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
318         // 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
319         // 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。
320         cli();
321         outb_p(14, video_port_reg);           // 选择数据寄存器 r14。
322         outb_p(0xff & ((pos-video_mem_start)>>9), video_port_val);
323         outb_p(15, video_port_reg);           // 选择数据寄存器 r15。
324         outb_p(0xff & ((pos-video_mem_start)>>1), video_port_val);
325         sti();
326     }
327
328     // 发送对 VT100 的响应序列。
329     // 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
330     // 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
331     // 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
332     // 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
333     // copy_to_cooked() 函数处理后放入辅助队列中。
334     static void respond(struct tty_struct * tty)
335     {
336         char * p = RESPONSE;                  // 定义在第 85 行上。
337
338         cli();
339         while (*p) {                          // 将应答序列放入读队列。
340             PUTCH(*p, tty->read_q);           // 逐字符放入。include/linux/tty.h, 33 行。
341             p++;
342         }
343         sti();                                // 转换成规范模式（放入辅助队列中）。
344         copy_to_cooked(tty);                  // tty_io.c, 145 行。
345     }
346
347     // 在光标处插入一空格字符。
348     // 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。
349     static void insert_char(void)
350     {
351         int i=x;
352         unsigned short tmp, old = video_erase_char; // 擦除字符（加属性）。
353         unsigned short * p = (unsigned short *) pos; // 光标对应内存位置。
354
355         while (i++<video_num_columns) {
356             tmp=*p;
357             *p=old;
358             old=tmp;
359             p++;
360         }

```

```

347     }
348 }
349
350 // 在光标处插入一行。
351 // 将屏幕窗口从光标所在行到窗口底的内容向下滚动一行。光标将处在新的空行上。
352 static void insert_line(void)
353 {
354     int oldtop, oldbottom;
355
356     // 首先保存屏幕窗口滚动开始行 top 和最后行 bottom 值，然后从光标所在行让屏幕内容向下
357     // 滚动一行。最后恢复屏幕窗口滚动开始行 top 和最后行 bottom 的原来值。
358     oldtop=top;
359     oldbottom=bottom;
360     top=y; // 设置屏幕滚动开始行和结束行。
361     bottom = video_num_lines;
362     scrdown(); // 从光标开始处，屏幕内容向下滚动一行。
363     top=oldtop;
364     bottom=oldbottom;
365 }
366
367 // 删除一个字符。
368 // 删除光标处的一个字符，光标右边的所有字符左移一格。
369 static void delete_char(void)
370 {
371     int i;
372     unsigned short * p = (unsigned short *) pos;
373
374     // 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
375     // 字符左移一格。然后在最后一个字符处填入擦除字符。
376     if (x>=video_num_columns)
377         return;
378     i = x;
379     while (++i < video_num_columns) { // 光标右所有字符左移 1 格。
380         *p = *(p+1);
381         p++;
382     }
383     *p = video_erase_char; // 最后填入擦除字符。
384 }
385
386 // 删除光标所在行。
387 // 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。
388 static void delete_line(void)
389 {
390     int oldtop, oldbottom;
391
392     // 首先保存屏幕滚动开始行 top 和最后行 bottom 值，然后从光标所在行让屏幕内容向上滚动
393     // 一行。最后恢复屏幕滚动开始行 top 和最后行 bottom 的原来值。
394     oldtop=top;
395     oldbottom=bottom;
396     top=y; // 设置屏幕滚动开始行和最后行。
397     bottom = video_num_lines;
398     scrup(); // 从光标开始处，屏幕内容向上滚动一行。
399     top=oldtop;

```

```

388         bottom=oldbottom;
389     }
390     ///// 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 n。
391 static void csi at(unsigned int nr)
392 {
    // 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。
393     if (nr > video num columns)
394         nr = video num columns;
395     else if (!nr)
396         nr = 1;
397     while (nr--)
398         insert char();
399 }
400     ///// 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。
401 static void csi L(unsigned int nr)
402 {
    // 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。
403     if (nr > video num lines)
404         nr = video num lines;
405     else if (!nr)
406         nr = 1;
407     while (nr--)
408         insert line();
409 }
410     ///// 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。
411 static void csi P(unsigned int nr)
412 {
    // 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。
413     if (nr > video num columns)
414         nr = video num columns;
415     else if (!nr)
416         nr = 1;
417     while (nr--)
418         delete char();
419 }

```



```

420
    ///// 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。
421 static void csi\_M(unsigned int nr)
422 {
    // 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。
423     if (nr > video\_num\_lines)
424         nr = video\_num\_lines;
425     else if (!nr)
426         nr=1;
427     while (nr--)
428         delete\_line();
429 }
430
431 static int saved\_x=0;           // 保存的光标列号。
432 static int saved\_y=0;           // 保存的光标行号。
433
    ///// 保存当前光标位置。
434 static void save\_cur(void)
435 {
436     saved\_x=x;
437     saved\_y=y;
438 }
439
    ///// 恢复保存的光标位置。
440 static void restore\_cur(void)
441 {
442     gotoxy(saved\_x, saved\_y);
443 }
444
    ///// 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。
445 void con\_write(struct tty\_struct * tty)
446 {
447     int nr;
448     char c;
449
    // 首先取得写缓冲队列中现有字符数 nr, 然后针对队列中的每个字符进行处理。在处理每个字
    // 符的循环过程中, 首先从写队列中取一字符 c, 根据前面处理字符所设置的状态 state 分步
    // 骤进行处理。状态之间的转换关系为:
    // state = 0: 初始状态, 或者原状态 4; 或者原状态 1, 但字符不是 '[';
    //      1: 原状态 0, 并且字符是转义字符 ESC (0x1b = 033 = 27)。处理后恢复状态 0。
    //      2: 原状态 1, 并且字符是 '[';
    //      3: 原状态 2, 或者原状态 3, 并且字符是 ';' 或数字。
    //      4: 原状态 3, 并且字符不是 ';' 或数字; 处理后恢复状态 0。
450     nr = CHARS(tty->write_q);
451     while (nr--) {

```

```

452         GETCH(tty->write_q, c);
453         switch(state) {
454             case 0:
// 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。
455                 if (c>31 && c<127) { // 是普通显示字符。
456                     if (x>video_num_columns) { // 要换行？
457                         x -= video_num_columns;
458                         pos -= video_size_row;
459                         lf();
460                     }
461                     __asm__( "movb _attr, %%ah\n\t" // 写字符。
462                             "movw %%ax, %l\n\t"
463                             ":: \"a\" (c), \"m\" (*(short *)pos)
464                             : \"ax\");
465                     pos += 2;
466                     x++;
// 如果字符 c 是转义字符 ESC，则转换状态 state 到 1。
467                     } else if (c==27) // ESC - 转义控制字符。
468                         state=1;
// 如果 c 是换行符 LF(10)，或垂直制表符 VT(11)，或换页符 FF(12)，则光标移动到下一行。
469                     else if (c==10 || c==11 || c==12)
470                         lf();
// 如果 c 是回车符 CR(13)，则将光标移动到头列(0 列)。
471                     else if (c==13) // CR - 回车。
472                         cr();
// 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。
473                     else if (c==ERASE_CHAR(tty))
474                         del();
// 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。
475                     else if (c==8) { // BS - 后退。
476                         if (x) {
477                             x--;
478                             pos -= 2;
479                         }
// 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。
480                     } else if (c==9) { // HT - 水平制表。
481                         c=8-(x&7);
482                         x += c;
483                         pos += c<<1;
484                         if (x>video_num_columns) {
485                             x -= video_num_columns;
486                             pos -= video_size_row;
487                             lf();
488                         }
489                         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。
490                     } else if (c==7)
491                         sysbeep();

```

```

492                                     break;
// 如果在原状态 0 收到转义字符 ESC (0x1b = 033 = 27), 则转到状态 1 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 0。
493                                     case 1:
494                                     state=0;
// 如果字符 c 是 '[' , 则将状态 state 转到 2。
495                                     if (c=='[')          // ESC [ - 是 CSI 序列。
496                                     state=2;
// 如果字符 c 是 'E' , 则光标移到下一行开始处 (0 列)。
497                                     else if (c=='E')      // ESC E - 光标下移 1 行回 0 列。
498                                     gotoxy(0, y+1);
// 如果字符 c 是 'M' , 则光标上移一行。
499                                     else if (c=='M')      // ESC M - 光标下移 1 行。
500                                     ri();
// 如果字符 c 是 'D' , 则光标下移一行。
501                                     else if (c=='D')      // ESC D - 光标下移 1 行。
502                                     lf();
// 如果字符 c 是 'Z' , 则发送终端应答字符序列。
503                                     else if (c=='Z')      // ESC Z - 设备属性查询。
504                                     respond(tty);
// 如果字符 c 是 '7' , 则保存当前光标位置。注意这里代码写错! 应该是 (c=='7')。
505                                     else if (x=='7')      // ESC 7 - 保存光标位置。
506                                     save_cur();
// 如果字符 c 是 '8' , 则恢复到原保存的光标位置。注意这里代码写错! 应该是 (c=='8')。
507                                     else if (x=='8')      // ESC 8 - 恢复保存的光标原位置。
508                                     restore_cur();
509                                     break;
// 如果在状态 1 (是转义字符 ESC) 时收到字符 '[' , 则表明是一个控制序列引导码 CSI, 于是
// 转到这里状态 2 来处理。首先对 ESC 转义字符序列保存参数的数组 par[] 清零, 索引变量
// npar 指向首项, 并且设置状态为 3。若此时字符不是 '7' , 则直接转到状态 3 去处理, 若此时
// 字符是 '7' , 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 3 处理代码处。否则直接进入状态 3 继续处理。
510                                     case 2:
511                                     for(npar=0; npar<NPAR; npar++)
512                                     par[npar]=0;
513                                     npar=0;
514                                     state=3;
515                                     if (ques==(c=='?'))
516                                     break;
// 状态 3 用于把转义字符序列中的数字字符转换成数值保存在 par[] 数组中。如果原是状态 2,
// 或者原来就是状态 3, 但原字符是 ';' 或数字, 则在状态 3 处理。此时, 如果字符 c 是分号
// ';' , 并且数组 par 未滿, 则索引值加 1, 准备处理下一个字符。
517                                     case 3:
518                                     if (c==';' && npar<NPAR-1) {
519                                     npar++;
520                                     break;
// 否则, 如果字符 c 是数字字符 '0'-'9' , 则将该字符转换成数值并与 npar 所索引的项组成
// 10 进制数, 并准备处理下一个字符。否则就直接转到状态 4。
521                                     } else if (c>='0' && c<='9') {
522                                     par[npar]=10*par[npar]+c-'0';
523                                     break;
524                                     } else state=4;
// 状态 4 是处理转义字符序列的最后一步。根据前面几个状态的处理我们已经获得了转义字符

```

```

// 序列的前几部分，现在根据参数字符串中最后一个字符（命令）来执行相关的操作。如果原
// 状态是状态 3，并且字符不是 ';' 或数字，则转到状态 4 处理。首先复位状态 state=0。
525         case 4:
526             state=0;
527             switch(c) {
// 如果 c 是 '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。
528                 case 'G': case ``: // CSI Pn G - 光标水平移动。
529                     if (par[0]) par[0]--;
530                     gotoxy(par[0], y);
531                     break;
// 如果字符 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。
532                 case 'A': // CSI Pn A - 光标上移。
533                     if (!par[0]) par[0]++;
534                     gotoxy(x, y-par[0]);
535                     break;
// 如果字符 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。
536                 case 'B': case 'e': // CSI Pn B - 光标下移。
537                     if (!par[0]) par[0]++;
538                     gotoxy(x, y+par[0]);
539                     break;
// 如果字符 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。
540                 case 'C': case 'a': // CSI Pn C - 光标右移。
541                     if (!par[0]) par[0]++;
542                     gotoxy(x+par[0], y);
543                     break;
// 如果字符 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。
544                 case 'D': // CSI Pn D - 光标左移。
545                     if (!par[0]) par[0]++;
546                     gotoxy(x-par[0], y);
547                     break;
// 如果字符 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。
548                 case 'E': // CSI Pn E - 光标下移回 0 列。
549                     if (!par[0]) par[0]++;
550                     gotoxy(0, y+par[0]);
551                     break;
// 如果字符 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。
552                 case 'F': // CSI Pn F - 光标上移回 0 列。
553                     if (!par[0]) par[0]++;
554                     gotoxy(0, y-par[0]);
555                     break;
// 如果字符 c 是 'd'，则在当前列设置行位置。第 1 个参数代表光标所需的行号（从 0 计数）。
556                 case 'd': // CSI Pn d - 在当前列置行位置。
557                     if (par[0]) par[0]--;
558                     gotoxy(x, par[0]);
559                     break;
// 如果字符 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。
560                 case 'H': case 'f': // CSI Pn H - 光标定位。
561                     if (par[0]) par[0]--;
562                     if (par[1]) par[1]--;
563                     gotoxy(par[1], par[0]);
564                     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：
// 转义序列：'ESC [sJ'（s=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）。

```

```

565         case 'J':           // CSI Pn J - 屏幕擦除字符。
566             csi_J(par[0]);
567             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 [sK' (s = 0 删除到行尾; 1 从开始删除; 2 整行都删除)。
568         case 'K':           // CSI Pn K - 行内擦除字符。
569             csi_K(par[0]);
570             break;
// 如果字符 c 是 'L', 表示在光标位置处插入 n 行 (转义序列 'ESC [ Pn L' )。
571         case 'L':           // CSI Pn L - 插入行。
572             csi_L(par[0]);
573             break;
// 如果字符 c 是 'M', 表示在光标位置处删除 n 行 (转义序列 'ESC [ Pn M' )。
574         case 'M':           // CSI Pn M - 删除行。
575             csi_M(par[0]);
576             break;
// 如果字符 c 是 'P', 表示在光标位置处删除 n 个字符 (转义序列 'ESC [ Pn P' )。
577         case 'P':           // CSI Pn P - 删除字符。
578             csi_P(par[0]);
579             break;
// 如果字符 c 是 '@', 表示在光标位置处插入 n 个字符 (转义序列 'ESC [ Pn @' )。
580         case '@':           // CSI Pn @ - 插入字符。
581             csi_at(par[0]);
582             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。
// ANSI 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。
583         case 'm':           // CSI Ps m - 设置显示字符属性。
584             csi_m();
585             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。
586         case 'r':           // CSI Pn;Pn r - 设置滚屏上下界。
587             if (par[0]) par[0]--;
588             if (!par[1]) par[1] = video_num_lines;
589             if (par[0] < par[1] &&
590                 par[1] <= video_num_lines) {
591                 top=par[0];
592                 bottom=par[1];
593             }
594             break;
// 如果字符 c 是 's', 则表示保存当前光标所在位置。
595         case 's':           // CSI s - 保存光标位置。
596             save_cur();
597             break;
// 如果字符 c 是 'u', 则表示恢复光标到原保存的位置处。
598         case 'u':           // CSI u - 恢复保存的光标位置。
599             restore_cur();
600             break;
601     }
602 }
603 }
604 set_cursor(); // 最后根据上面设置的光标位置, 向显示控制器发送光标显示位置。
605 }
606

```

```

607 /*
608  * void con_init(void);
609  *
610  * This routine initializes console interrupts, and does nothing
611  * else. If you want the screen to clear, call tty_write with
612  * the appropriate escape-sequence.
613  *
614  * Reads the information preserved by setup.s to determine the current display
615  * type and sets everything accordingly.
616  */
/*
 * 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。
 */
///// 控制台初始化程序。在 init/main.c 中被调用。
// 该函数首先根据 setup.s 程序取得的系统硬件参数初始化设置几个本函数专用的静态全局变
// 量。然后根据显示卡模式（单色还是彩色显示）和显示卡类型（EGA/VGA 还是 CGA）分别设
// 置显示内存起始位置以及显示索引寄存器和显示数值寄存器端口号。最后设置键盘中断陷阱
// 描述符并复位对键盘中断的屏蔽位，以允许键盘开始工作。
// 第 619 行定义一个寄存器变量 a，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（如 eax），则可写成“register unsigned char a asm("ax");”。
617 void con_init(void)
618 {
619     register unsigned char a;
620     char *display_desc = "????";
621     char *display_ptr;
622
623     // 首先根据 setup.s 程序取得的系统硬件参数（见本程序第 39—47 行）初始化几个本函数专用
624     // 的静态全局变量。
625     video_num_columns = ORIG_VIDEO_COLS; // 显示器显示字符列数。
626     video_size_row = video_num_columns * 2; // 每行字符需使用的字节数。
627     video_num_lines = ORIG_VIDEO_LINES; // 显示器显示字符行数。
628     video_page = ORIG_VIDEO_PAGE; // 当前显示页面。
629     video_erase_char = 0x0720; // 擦除字符（0x20 是字符，0x07 属性）。
630
631     // 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
632     // 端口号和显示寄存器数据端口号。如果原始显示模式等于 7，则表示是单色显示器。
633     if (ORIG_VIDEO_MODE == 7) /* Is this a monochrome display? */
634     {
635         video_mem_start = 0xb0000; // 设置单显映像内存起始地址。
636         video_port_reg = 0x3b4; // 设置单显索引寄存器端口。
637         video_port_val = 0x3b5; // 设置单显数据寄存器端口。
638
639         // 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
640         // 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
641         // 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
642         // 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'。
643         // 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。
644         // 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
645         // 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。
634         if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
635         {
636             video_type = VIDEO_TYPE_EGAM; // 设置显示类型(EGA 单色)。
637             video_mem_end = 0xb8000; // 设置显示内存末端地址。
638             display_desc = "EGAm"; // 设置显示描述字符串。
639         }
// 如果 BX 寄存器的值等于 0x10，则说明是单色显示卡 MDA。则设置相应参数。
640     else
641     {
642         video_type = VIDEO_TYPE_MDA; // 设置显示类型(MDA 单色)。
643         video_mem_end = 0xb2000; // 设置显示内存末端地址。
644         display_desc = "MDA"; // 设置显示描述字符串。
645     }
646 }
// 如果显示模式不为 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。
647     else /* If not, it is color. */
648     {
649         video_mem_start = 0xb8000; // 显示内存起始地址。
650         video_port_reg = 0x3d4; // 设置彩色显示索引寄存器端口。
651         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/VGA 显示卡。此时实际上我们可以使
// 用 32KB 显示内存 (0xb8000 -- 0xc0000)，但该程序只使用了其中 16KB 显示内存。
652         if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
653         {
654             video_type = VIDEO_TYPE_EGAC; // 设置显示类型(EGA 彩色)。
655             video_mem_end = 0xbc000; // 设置显示内存末端地址。
656             display_desc = "EGAc"; // 设置显示描述字符串。
657         }
// 如果 BX 寄存器的值等于 0x10，则说明是 CGA 显示卡。只使用 8KB 显示内存。
658     else
659     {
660         video_type = VIDEO_TYPE_CGA; // 设置显示类型(CGA)。
661         video_mem_end = 0xba000; // 设置显示内存末端地址。
662         display_desc = "CGA"; // 设置显示描述字符串。
663     }
664 }
665
666 /* Let the user known what kind of display driver we are using */
667 /* 让用户知道我们正在使用哪一类显示驱动程序 */
668
// 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。
668     display_ptr = ((char *)video_mem_start) + video_size_row - 8;
669     while (*display_desc)
670     {
671         *display_ptr++ = *display_desc++; // 复制字符。
672         display_ptr++; // 空开属性字节位置。
673     }
674
675 /* 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） */
676
677 origin = video_mem_start;           // 滚屏起始显示内存地址。
678 scr_end = video_mem_start + video_num_lines * video_size_row; // 结束地址。
679 top = 0;                             // 最顶行号。
680 bottom = video_num_lines;           // 最底行号。
681
// 最后初始化当前光标所在位置和光标对应的内存位置 pos。并设置设置键盘中断 0x21 陷阱门
// 描述符，&keyboard_interrupt 是键盘中断处理过程地址。取消 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。
682 gotoxy(ORIG_X, ORIG_Y);
683 set_trap_gate(0x21, &keyboard_interrupt); // 参见 system.h，第 36 行开始。
684 outb_p(inb_p(0x21) & 0xfd, 0x21); // 取消对键盘中断的屏蔽，允许 IRQ1。
685 a = inb_p(0x61); // 读取键盘端口 0x61（8255A 端口 PB）。
686 outb_p(a | 0x80, 0x61); // 设置禁止键盘工作（位 7 置位），
687 outb(a, 0x61); // 再允许键盘工作，用以复位键盘。
688 }
689 /* from bsd-net-2: */
690
///// 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。
691 void sysbeepstop(void)
692 {
693     /* disable counter 2 */ /* 禁止定时器 2 */
694     outb(inb_p(0x61) & 0xFC, 0x61);
695 }
696
697 int beepcount = 0; // 蜂鸣时间嘀嗒计数。
698
// 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。
699 static void sysbeep(void)
700 {
701     /* enable counter 2 */ /* 开启定时器 2 */
702     outb_p(inb_p(0x61) | 3, 0x61);
703     /* set command for counter 2, 2 byte write */ /* 送设置定时器 2 命令 */
704     outb_p(0xB6, 0x43); // 定时器芯片控制字寄存器端口。
705     /* send 0x637 for 750 HZ */ /* 设置频率为 750HZ，因此送定时值 0x637 */
706     outb_p(0x37, 0x42); // 通道 2 数据端口分别送计数高低字节。
707     outb(0x06, 0x42);
708     /* 1/8 second */ /* 蜂鸣时间为 1/8 秒 */
709     beepcount = HZ/8;
710 }
711

```

10.4.3 其他信息

10.4.3.1 显示控制卡编程

这里仅给出和说明兼容显示卡端口的说明。描述了 MDA、CGA、EGA 和 VGA 显示控制卡的通用

编程端口，这些端口都是与 CGA 使用的 MC6845 芯片兼容，其名称和用途见表 10-5 所示。其中以 CGA/EGA/VGA 的端口(0x3d0-0x3df)为例进行说明，MDA 的端口是 0x3b0 - 0x3bf。

对显示控制卡进行编程的基本步骤是：首先把 0--17 值写入显示卡的索引寄存器（端口 0x3d4），选择要进行设置的显示控制内部寄存器之一(r0-r17)，此时数据寄存器端口（0x3d5）对应到该内部寄存器上。然后将参数写到该数据寄存器端口。也即显示卡的数据寄存器端口每次只能对显示卡中的一个内部寄存器进行操作。内部寄存器见表 10-6 所示。

表 10-5 CGA 端口寄存器名称及作用

端口	读/写	名称和用途
0x3d4	写	CRT(6845)索引寄存器。用于选择通过端口 0x3d5 访问的各个数据寄存器(r0-r17)。
0x3d5	写	CRT(6845)数据寄存器。其中数据寄存器 r14-r17 还可以读。 各个数据寄存器的功能说明见表 10-6。
0x3d8	读/写	模式控制寄存器。 位 7-6 未用； 位 5=1 允许闪烁； 位 4=1 640*200 图形模式； 位 3=1 允许视频； 位 2=1 单色显示； 位 1=1 图形模式；=0 文本模式； 位 0=1 80*25 文本模式；=0 40*25 文本模式。
0x3d9	读/写	CGA 调色板寄存器。选择所采用的色彩。 位 7-6 未用； 位 5=1 激活色彩集：青(cyan)、紫(magenta)、白(white)； =0 激活色彩集：红(red)、绿(green)、蓝(blue)； 位 4=1 增强显示图形、文本背景色彩； 位 3=1 增强显示 40*25 的边框、320*200 的背景、640*200 的前景颜色； 位 2=1 显示红色：40*25 的边框、320*200 的背景、640*200 的前景； 位 1=1 显示绿色：40*25 的边框、320*200 的背景、640*200 的前景； 位 0=1 显示蓝色：40*25 的边框、320*200 的背景、640*200 的前景；
0x3da	读	CGA 显示状态寄存器。 位 7-4 未用； 位 3=1 在垂直回扫阶段； 位 2=1 光笔开关关闭；=0 光笔开关接通； 位 1=1 光笔选通有效； 位 0=1 可以干扰显示访问显示内存；=0 此时不要使用显示内存。
0x3db	写	清除光笔锁存（复位光笔寄存器）。
0x3dc	读/写	预设置光笔锁存（强制光笔选通有效）。

表 10-6 MC6845 内部数据寄存器及初始值

编号	名称	单位	读/写	40*25 模式	80*25 模式	图形模式
r0	水平字符总数	字符	写	0x38	0x71	0x38
r1	水平显示字符数	字符	写	0x28	0x50	0x28
r2	水平同步位置	字符	写	0x2d	0x5a	0x2d

r3	水平同步脉冲宽度	字符	写	0x0a	0x0a	0x0a
r4	垂直字符总数	字符行	写	0x1f	0x1f	0x7f
r5	垂直同步脉冲宽度	扫描行	写	0x06	0x06	0x06
r6	垂直显示字符数	字符行	写	0x19	0x19	0x64
r7	垂直同步位置	字符行	写	0x1c	0x1c	0x70
r8	隔行/逐行选择		写	0x02	0x02	0x02
r9	最大扫描行数	扫描行	写	0x07	0x07	0x01
r10	光标开始位置	扫描行	写	0x06	0x06	0x06
r11	光标结束位置	扫描行	写	0x07	0x07	0x07
r12	显示内存起始位置(高)		写	0x00	0x00	0x00
r13	显示内存起始位置(低)		写	0x00	0x00	0x00
r14	光标当前位置(高)		读/写	可变		
r15	光标当前位置(低)		读/写			
r16	光笔当前位置(高)		读	可变		
r17	光笔当前位置(低)		读			

10.4.3.2 滚屏操作原理

滚屏操作是指将指定开始行和结束行的一块文本内容向上移动(向上卷动 scroll up)或向下移动(向下卷动 scroll down), 如果将屏幕看作是显示内存上对应屏幕内容的一个窗口的话, 那么将屏幕内容向上移即是窗口沿显示内存向下移动; 将屏幕内容向下移动即是窗口向上移动。在程序中就是重新设置显示控制器中显示内存的起始位置 `origin` 以及调整程序中相应的变量。对于这两种操作各自都有两种情况。

对于向上卷动, 当屏幕对应的显示内存窗口在向下移动后仍然在显示内存范围之内, 也即对应当前屏幕的内存块位置始终在显示内存起始位置(`video_mem_start`)和末端位置 `video_mem_end` 之间, 那么只需要调整显示控制器中起始显示内存位置即可。但是当对应屏幕的内存块位置在向下移动时超出了实际显示内存的末端(`video_mem_end`)这种情况, 就需要移动对应显示内存中的数据, 以保证所有当前屏幕数据都落在显示内存范围内。在这第二种情况, 程序中是将屏幕对应的内存数据移动到实际显示内存的开始位置处(`video_mem_start`)。

程序中实际的处理过程分三步进行。首先调整屏幕显示起始位置 `origin`; 然后判断对应屏幕内存数据是否超出显示内存下界(`video_mem_end`), 如果超出就将屏幕对应的内存数据移动到实际显示内存的开始位置处(`video_mem_start`); 最后对移动后屏幕上出现的新行用空格字符填满。见图 10-8 所示。其中图 (a)对应第一种简单情况, 图(b)对应需要移动内存数据时的情况。

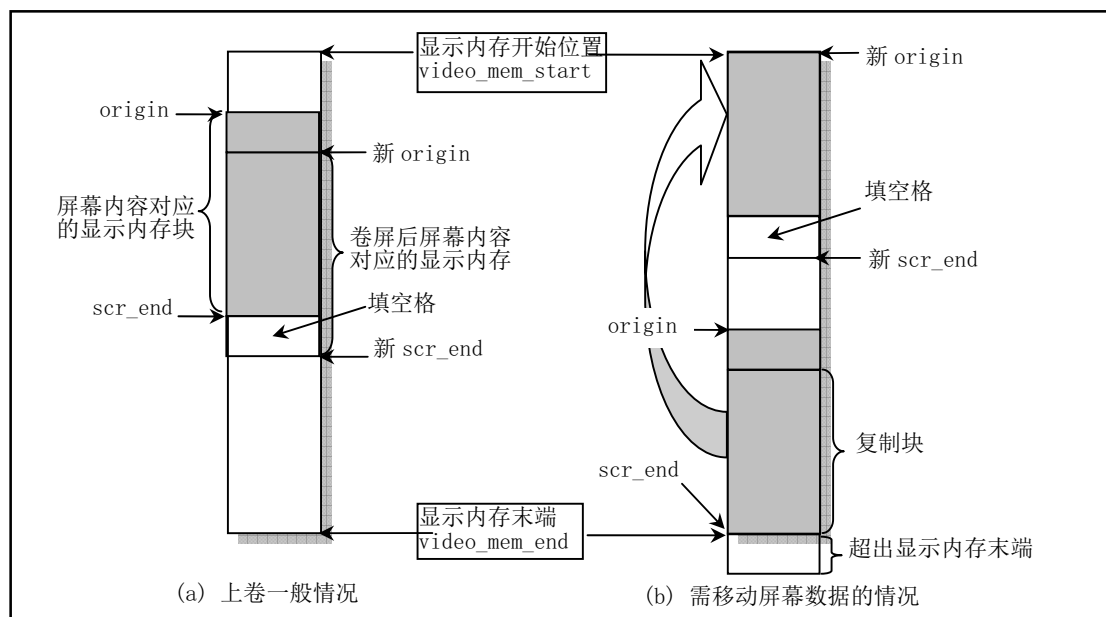


图 10-8 向上卷屏(scroll up)操作示意图

向下卷动屏幕的操作与向上卷屏相似，也会遇到这两种类似情况，只是由于屏幕窗口上移，因此会在屏幕上方出现一空行，并且在屏幕内容所对应的内存超出显示内存范围时需要将屏幕数据内存块往下移动到显示内存的末端位置。

10.4.3.3 终端控制命令

终端通常有两部分功能，分别作为计算机信息的输入设备(键盘)和输出设置(显示器)。终端可有许多控制命令，使得终端执行一定的操作而不是仅仅在屏幕上显示一个字符。使用这种方式，计算机就可以命令终端执行移动光标、切换显示模式和响铃等操作。终端控制命令又可分为两类：控制字符命令和 ANSI 转义控制序列。前面我们已经简单讨论过，Linux 内核中的 console.c（包括上面的 keyboard.s）程序实际上可以看作是模拟终端仿真程序。因此为了能理解 console.c 程序的处理过程，我们概要介绍一下一个终端设备中 ROM 中的程序如何处理从主机上接收到的代码数据。我们首先简单描述 ASCII 代码表结构，然后说明终端设备如何处理接收到的控制字符和控制序列字符串代码。

1. 字符编码方法

传统字符终端使用 ANSI(American National Standards Institute, 美国国家标准局)和 ISO(International Organization for Standardization, 国际标准化组织)标准的 8 比特编码方案和 7 比特代码扩展技术。ANSI 和 ISO 规定了计算机和通信领域的字符编码标准。ANSI X3.4-1977 和 ISO 646-1977 标准制定了美国信息交换处理代码集，即 ASCII 代码集。ANSI X3.41-1974 和 ISO 2022.2 标准描述了 7 比特和 8 比特编码集的代码扩展技术。ANSI X3.32、ANSI X3.64-1979 制定了利用 ASCII 码的文本字符表示终端控制字符的方法。虽然 Linux 0.1x 内核中仅实现对数字设备公司 DEC(现已纳入 Compaq 公司及 HP 公司)的 VT100 以及 VT102 终端设备的兼容，并且这两种事实上的标准终端设备仅支持 7 比特编码方案，但为了介绍的完整性和描述起来方便，这里我们仍然也同时介绍 8 比特编码方案。

2. 代码表

ASCII 码有 7 比特和 8 比特两种编码表示。7 比特代码表共有 128 个字符代码，见表 10-7 中左半部分所示。其中每行表示 7 比特中低 4 比特的值，而每列是高 3 比特值。例如第 4 列第 1 行代码'A'的 8 进制值是 0101，十进制值是 65 (0x41)。

表中的字符被分为两种类型。一种是第 1、第 2 列构成的控制字符(Control characters)，其余的是

图形字符（Graphic characters）或称为显示字符、文本字符。终端在接收到这两类字符时将分别进行处理。图形字符是可以在屏幕上显示的字符，而控制字符则通常不会在屏幕上显示。控制字符用于在数据通信和文本处理过程中起特殊的控制作用。另外，DEL 字符(0x7F)也是一个控制字符，而空格字符(0x20)既可以是一般文本字符也可以作为一个控制字符使用。控制字符及其功能已由 ANSI 标准化，其中的名称是 ANSI 标准的助记符。例如：CR（Carriage Return，回车符）、FF（Form Feed，换页符）和 CAN（Cancel，取消符）。通常 7 比特编码方式也适用于 8 比特的编码。表 10-7 是 8 比特代码表（其中左半个表与 7 比特代码表完全相同），其中右半部分的扩展代码没有列出。

表 10-7 8 比特 ASCII 代码表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p			无					
1	SOH	DC1	!	1	A	Q	a	q								
2	STX	DC2	"	2	B	R	b	r								
3	ETX	DC3	#	3	C	S	c	s								
4	EOT	DC4	\$	4	D	T	d	t	IND							
5	ENQ	NAK	%	5	E	U	e	u	NEL							
6	ACK	SYN	&	6	F	V	f	v	SSA							
7	BEL	ETB	'	7	G	W	g	w	ESA							
8	BS	CAN	(8	H	X	h	x	HTS							
9	HT	EM)	9	I	Y	i	y	HTJ							
A	LF	SUB	*	:	J	Z	j	z	VTB							
B	VT	ESC	+	;	K	[k	{	PLD	CSI						
C	FF	FS	,	<	L	\	l		PLU	ST						
D	CR	GS	-	=	M]	m	}	RI	OSC						
E	SO	RS	.	>	N	^	n	~	SS2	PM						
F	SI	US	/	?	O	_	o	DE L	SS3	APC						无
	C0 代码区		GL 代码区						C1 代码区		GR 代码区					
	7 比特代码表								8 比特代码表右半部分							

它比 7 比特代码表多出 8 列代码，共含有 256 个代码值。类似于 7 比特代码表，它每行代表 8 比特代码的低 4 比特值，而每列表示高 4 比特值。左面半个表（列 0--列 7）与 7 比特代码表完全一样，它们代码的第 8 比特为 0，因此该比特可以忽略。右面半个表（列 8--列 15）中各代码的第 8 比特均为 1，因此这些字符只能在 8 比特环境中使用。8 比特代码表有两个控制代码集：C0 和 C1。同时也有两个图形字符集：左图形字符集 GL（Graphic Left）和右图形字符集 GR（Graphic Right）。

C0 和 C1 中控制字符的功能不能更改，但是我们可以把不同的显示字符映射到 GL 和/或 GR 区域中。能够使用（映射）的各种文本字符集通常储存在终端设备中。在使用它们之前我们必须首先作映射操作。对于已成为事实上标准的 DEC 终端设备来说，其中通常储存有 DEC 多国字符集（ASCII 字符集和 DEC 辅助字符集）、DEC 特殊字符集和国家替换字符集 NCR（National Replacement Character）。当打开终端设备时，默认使用的就是 DEC 多国字符集。

3. 控制功能

为了指挥终端设备如何处理接收到的数据，我们就要使用终端设备的控制功能。主机通过发送控制代码或控制代码序列就可以控制终端设备对字符的显示处理，它们仅用作控制文本字符的显示、处理和

传送，而其本身并不显示在屏幕上。控制功能有许多用途，例如：在显示屏上移动光标位置、删除一行文本、更改字符、更改字符集和设置终端操作模式等。我们可以在文本模式中使用所有的控制功能，并用一个字节或多个字节来表示控制功能。

可以认为所有不用作在屏幕上显示的控制字符或控制字符序列都是控制功能。在每个符合 ANSI 标准的终端设备中并不是所有控制功能都能执行其控制操作，但是设备应该能够识别所有的控制功能，并忽略其中不起作用的控制功能。所以通常一个终端设备仅实现 ANSI 控制功能的一个子集。由于各种不同的设备使用不同的控制功能子集，因此与 ANSI 标准兼容并不意味着这些设备互相兼容。兼容性仅体现在各种设备使用相同的控制功能方面。

单字节控制功能就是 C0 和 C1 中的控制字符。使用 C0 中的控制字符可以获得有限的控制功能。而 C1 中的控制字符可以另外再提供一些控制功能，但只能在 8 比特环境中直接使用，因此 Linux 内核中所仿真的 VT100 型终端仅能使用 C0 中的控制字符。多字节控制代码则可以提供很多的控制功能。这些多字节控制代码通常被称为转义序列 (Escape Sequences)、控制序列 (Control Sequences) 和设备控制字符串 (Device Control Strings)。其中有些控制序列是工业界通用的 ANSI 标准序列，另外还有一些则是生产商为自己产品使用而设计的专有控制序列。象 ANSI 标准序列一样，专有控制序列字符也符合 ANSI 字符代码的组合标准。

4. 转义序列

主机可以发送转义序列来控制终端屏幕上文本字符的显示位置和属性。转义序列 (Escape Sequences) 由 C0 中控制字符 ESC (0x1b) 开始，后面跟随一个或多个 ASCII 显示字符。转义序列的 ANSI 标准格式见如下所示：

ESC	I. I	F
0x1b	0x20--0x2f	0x30--0x7e
引导码	中间字符	结尾字符
	0 或多个字符	1 个字符

ESC 是 ANSI 标准中定义的转义序列引导码 (Escape Sequence Introducer)。在接收到引导码 ESC 之后，终端需要以一定的顺序保存 (而非显示) 随后所有的控制字符。

中间字符 (Intermediate Characters) 是 ESC 之后接收到的范围在 0x20 -- 0x2f (ASCII 表中列 2) 的字符。终端需要把它们作为控制功能的一部分保存下来。

结尾字符 (Final Character) 是 ESC 之后接收到的范围在 0x30 -- 0x7e (ASCII 表中列 3 -- 7) 的字符。结尾字符指明转义序列的结束。中间字符和结尾字符共同定义了一个序列的功能。此时终端即可以执行指定的功能并继续显示随后收到的字符。ANSI 标准转义序列的结尾字符范围在 0x40 -- 0x7e (ASCII 表中列 4 -- 7)。各个终端设备厂家自己定义的专有转义序列的结尾字符范围在 0x30 -- 0x3f (ASCII 表中列 3)。例如下面序列就是一个用来指定 G0 作为 ASCII 字符集的转义序列：

ESC	(B
0x1b	0x28	0x42

由于转义序列仅使用 7 比特字符，因此我们可以在 7 比特和 8 比特环境中使用它们。请注意，当使用转义或控制序列时，要记得它们定义了一个代码序列而非字符的文本表示。这里这些字符仅用作体现可读性。转义序列的重要用途之一是扩展 7 比特控制字符的功能。ANSI 标准允许我们使用 2 字节转义序列作为 7 比特代码扩展来表示 C1 中的任何控制字符。在需要兼容 7 比特的应用环境中，这是一个非常有用的特性。例如，C1 中的控制字符 CSI 和 IND 可以使用 7 比特代码扩展形式象下面这样来表示：

C1 字符	转义序列
CSI	ESC [
0x9b	0x1b 0x5b
IND	ESC D
0x84	0x1b 0x44

通常，我们可以在两方面使用上述代码扩展技术。我们可以使用 2 字符转义序列来表示 8 比特代码表 C1 中的任何控制字符。其中第 2 个字符的值是 C1 中对应字符的值减 0x40 (64)。另外，我们也可以通过删去控制字符 ESC 并给第 2 个字符加上 0x40，把第 2 个字符值在 0x40 -- 0x5f 之间的任何转义序列转换成产生一个 8 比特的控制字符。

5. 控制序列

控制序列 (Control Sequences) 由控制字符 CSI (0x9b) 开始，后面跟随 1 个或多个 ASCII 图形字符。控制序列的 ANSI 标准格式见如下所示：

CSI	P.....P	I.....I	F
0x9b	0x30--0x3f	0x20--0x2f	0x40--0x7e
引导码	参数字符	中间字符	结尾字符
	0 或多个字符	0 或多个字符	1 个字符

控制序列引导码 (Control Sequence Introducer) 是控制字符 C1 中的 CSI (0x9b)。但由于 CSI 也可以使用 7 比特代码扩展 'ESC [' 来表示，因此所有控制序列都可以利用第 2 个字符是左方括号 '[' 的转义序列来表示。在接收到引导码 CSI 之后，终端需要以一定的顺序保存 (而非显示) 随后所有的控制字符。

参数字符 (Parameter Characters) 是 CSI 之后接收到的范围在 0x30 -- 0x3f (ASCII 表中列 3) 的字符。参数字符用于修改控制序列的作用或含义。当参数字符以任一 '< = > ?' (0x3c -- 0x3f) 字符开头时，终端将把本控制序列作为专有 (私有) 控制序列。终端可使用两类参数字符：数字字符和选择字符。数字字符参数代表一个十进制数，用 Pn 表示。范围是 0 -- 9。选择字符参数来自于一个指定的参数表，用 Ps 表示。如果一个控制序列中包含不止一个参数，则用分号 ';' (0x3b) 来隔开。

中间字符 (Intermediate Characters) 是 CSI 之后接收到的范围在 0x20 -- 0x2f (ASCII 表中列 2) 的字符。终端需要把它们作为控制功能的一部分保存下来。注意，终端设备不使用中间字符。

结尾字符 (Final Character) 是 CSI 之后接收到的范围在 0x40 -- 0x7e (ASCII 表中列 4 -- 7) 的字符。结尾字符指明控制序列的结束。中间字符和结尾字符共同定义了一个序列的功能。此时终端即可以执行指定的功能并继续显示随后收到的字符。ANSI 标准转义序列的结尾字符范围在 0x40 -- 0x6f (ASCII 表中列 4 -- 6)。各个终端设备厂家自己定义的专有转义序列的结尾字符范围在 0x70 -- 0x7e (ASCII 表中列 7)。例如，下面序列定义了一个使屏幕光标移动到指定位置 (行 5、列 9) 的控制序列：

CSI	5	;	9	H	
0x9b	0x35	0x3b	0x41	0x48	
或者:					
ESC	[5	;	9	H
0x1b	0x5b	0x35	0x3b	0x39	0x48

图 10-9 中是一个控制序列的例子：取消所有字符的属性，然后开启下划线和反显属性。ESC [0;4;7m

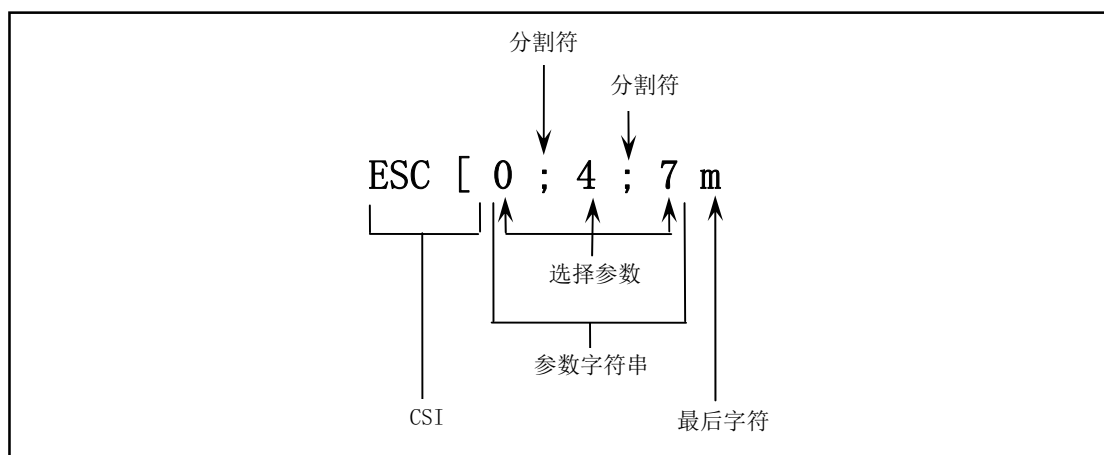


图 10-9 控制序列例子

6. 终端对接收到代码的处理

本节说明终端如何处理接收到的字符，即描述终端对从应用程序或主机系统接收到的代码的响应。接收到的字符可分为两类：图形（显示或文本）字符和控制字符。图形字符是接收到的显示在屏幕上的字符。实际在屏幕上显示的字符依赖于所选择的字符集。字符集可通过控制功能来选择。

终端收到的所有数据由一个或多个字符代码组成。这些数据包括图形字符、控制字符、转义序列、控制序列以及设备控制串。绝大多数数据是由仅在屏幕上显示的图形字符构成，并没有其他作用。控制字符、转义序列、控制序列以及设备控制串都是“控制功能”，我们可以在自己的程序或操作系统中使用它们来指明终端如何处理、传送和显示字符。每个控制功能有一个唯一的名称，并且都有一个简写助记符。这些名称和助记符都已成为标准。默认情况下，终端对某个控制或显示字符的解释依赖于 ASCII 码字符集。

注意：对于不支持的控制代码，终端通常采取的操作是忽略它。随后发送到终端的不是这里说明的字符有可能会造成不可预测的后果。

本书附录中给出了常用的 C0 和 C1 表中控制字符的说明，概要描述了当终端收到会采取的操作。对于一个特定的终端，它通常并不会识别 C0 和 C1 中所有的控制字符。另外，附录中还用表的形式列出了 Linux 0.1x 内核中 console.c 程序使用的转义序列和控制序列。除特别说明以外，所有序列均表示主机发送过来的控制功能序列。

10.5 serial.c 程序

10.5.1 功能描述

本程序实现系统串行端口初始化，为使用串行终端设备作好准备工作。在 rs_init() 初始化函数中，设置了默认的串行通信参数，并设置串行端口的中断陷阱门（中断向量）。rs_write() 函数用于把串行终端设备写缓冲队列中的字符通过串行线路发送给远端的终端设备。

rs_write() 将在文件系统中用于操作字符设备文件时被调用。当一个程序往串行设备/dev/tty64 文件执行写操作时，就会执行系统调用 sys_write()（在 fs/read_write.c 中），而这个系统调用在判别出所读文件是一个字符设备文件时，即会调用 rw_char() 函数（在 fs/char_dev.c 中），该函数则会根据所读设备的子设备号等信息，由字符设备读写函数表（设备开关表）调用 rw_tty()，最终调用到这里的串行终端写操作函数 rs_write()。

rs_write() 函数实际上只是开启串行发送保持寄存器已空中断标志，在 UART 将数据发送出去后允许

发中断信号。具体发送操作是在 rs_io.s 程序中完成。

10.5.2 代码注释

程序 10-4 linux/kernel/chr_drv/serial.c

```

1  /*
2   *  linux/kernel/serial.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *      serial.c
9   *
10 *  This module implements the rs232 io functions
11 *      void rs_write(struct tty_struct * queue);
12 *      void rs_init(void);
13 *  and all interrupts pertaining to serial IO.
14 */
15
16 /*
17 *      serial.c
18 *  该程序用于实现 rs232 的输入输出函数
19 *      void rs_write(struct tty_struct *queue);
20 *      void rs_init(void);
21 *  以及与串行 IO 有关系的所有中断处理程序。
22 */
23
24 #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。
25 #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。
26 #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。
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
29 #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。
30
31 extern void rs1_interrupt(void);        // 串行口 1 的中断处理程序(rs_io.s, 34)。
32 extern void rs2_interrupt(void);        // 串行口 2 的中断处理程序(rs_io.s, 38)。
33
34
35 // 初始化串行端口
36 // 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。
37 // 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)。
38 // 参数: port 是串行端口基地址, 串口 1 - 0x3F8; 串口 2 - 0x2F8。
39
40 static void init(int port)
41 {
42     outb_p(0x80, port+3);    /* set DLAB of line control reg */
43                             /* 设置线路控制寄存器的 DLAB 位(位 7) */
44     outb_p(0x30, port);      /* LS of divisor (48 -> 2400 bps */
45                             /* 发送波特率因子低字节, 0x30->2400bps */
46     outb_p(0x00, port+1);    /* MS of divisor */
47                             /* 发送波特率因子高字节, 0x00 */
48     outb_p(0x03, port+3);    /* reset DLAB */
49                             /* 复位 DLAB 位, 数据位为 8 位 */
50     outb_p(0x0b, port+4);    /* set DTR, RTS, OUT_2 */
51 }

```

```

33         outb_p(0x0d, port+1);    /* 设置 DTR, RTS, 辅助用户输出 2 */
34         (void)inb(port);          /* enable all intrs but writes */
35     }                               /* 除了写(写保持空)以外, 允许所有中断源中断 */
36                                     /* read data port to reset things (?) */
37                                     /* 读数据口, 以进行复位操作(?) */
38     }
39     // 初始化串行中断程序和串行接口。
40     // 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。
41     void rs_init(void)
42     {
43         // 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。
44         // 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。
45         set_intr_gate(0x24, rs1_interrupt); // 设置串口 1 的中断门向量 (IRQ4 信号)。
46         set_intr_gate(0x23, rs2_interrupt); // 设置串口 2 的中断门向量 (IRQ3 信号)。
47         init(tty_table[1].read_q.data);    // 初始化串口 1 (.data 是端口基地址)。
48         init(tty_table[2].read_q.data);    // 初始化串口 2。
49         outb(inb_p(0x21)&0xE7, 0x21);      // 允许主 8259A 响应 IRQ3、IRQ4 中断请求。
50     }
51
52     /*
53     * This routine gets called when tty_write has put something into
54     * the write_queue. It must check wheter the queue is empty, and
55     * set the interrupt register accordingly
56     *
57     * void _rs_write(struct tty_struct * tty);
58     */
59     /*
60     * 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
61     * 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。
62     */
63     // 串行数据发送输出。
64     // 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
65     // 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
66     // 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。
67     // 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
68     // 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
69     // 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”
70     // 发送操作也随之结束。
71     void rs_write(struct tty_struct * tty)
72     {
73         cli(); // 关中断。
74         // 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
75         // 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
76         // 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。
77         if (!EMPTY(tty->write_q))
78             outb(inb_p(tty->write_q.data+1)|0x02, tty->write_q.data+1);
79         sti(); // 开中断。
80     }
81
82 
```

10.5.3 其他信息

10.5.3.1 异步串行通信控制器 UART

异步串行通信传输的帧格式见图 10-10 所示。传输一个字符由起始位、数据位、奇偶校验位和停止位构成。其中起始位起同步作用，值恒为 0。数据位是传输的实际数据，即一个字符的代码。其长度可以是 5--8 个比特。奇偶校验位可有可无，由程序设定。停止位恒为 1，可由程序设定为 1、1.5 或 2 个比特。在通信开始发送信息之前，双方必须设置成相同的格式。如具有相同数量的数据比特位和停止位。在异步通信规范中，把传送 1 称为传号（MARK），传送 0 称为空号（SPACE）。因此在下面描述中我们就使用这两个术语。

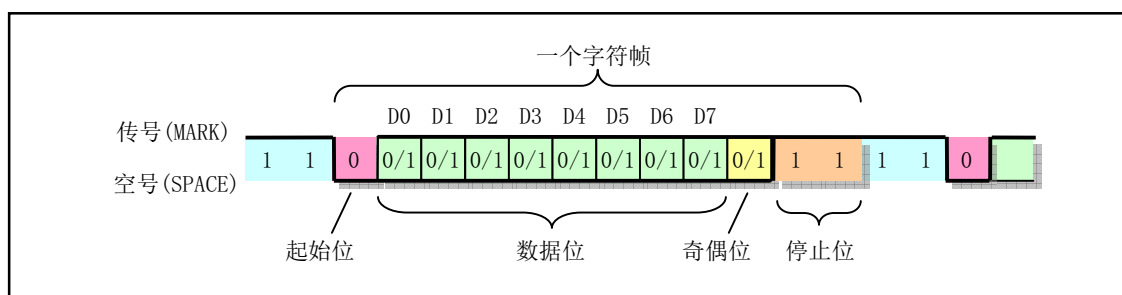


图 10-10 异步串行通信字符传输格式

当无数据传输时，发送方处于传号（MARK）状态，持续发送 1。若需要发送数据，则发送方需要首先发送一个比特位间隔时间的空号起始位。接收方收到空号后，就开始与发送方同步，然后接收随后的数据。若程序中设置了奇偶校验位，那么在数据传输完之后还需要接收奇偶校验位。最后是停止位。在一个字符帧发送完后可以立刻发送下一个字符帧，也可以暂时发送传号，等一会再发送字符帧。

在接收一字符帧时，接收方可能会检测到三种错误之一：①奇偶校验错误。此时程序应该要求对方重新发送该字符；②过速错误。由于程序取字符速度慢于接收速度，就会发生这种错误。此时应该修改程序加快取字符频率；③帧格式错误。在要求接收的格式信息不正确时会发生这种错误。例如在应该收到停止位时却收到了空号。通常造成这种错误的情况除了线路干扰以外，很可能是通信双方的帧格式设置的不同。

1. 串行通信接口及 UART 结构

为实现串行通信，PC 机上通常都带有 2 个符合 RS-232C 标准的串行接口，并使用通用异步接收/发送器控制芯片 UART（Universal Asynchronous Receiver/Transmitter）来处理串行数据的收发工作。PC 机上的串行接口通常使用 25 芯或 9 芯的 DB-25 或 DB-9 连接器，主要用来连接 MODEM 设备进行工作，因此 RS-232C 标准规定了很多 MODEM 专用接口引线。有关 RS-232C 标准和 MODEM 设备工作原理的详细说明请参考其它资料。这里我们主要说明 UART 控制芯片的结构。

以前的 PC 机都使用国家半导体公司的 NS8250 或 NS16450 UART 芯片，现在的 PC 机则使用了 16650A 及其兼容芯片，但都与 NS8250/16450 芯片兼容。NS8250/16450 与 16650A 芯片的主要区别在于 16650A 芯片还另外支持 FIFO 传输方式。在这种方式下，UART 可以在接收或发送了最多 16 个字符后才引发一次中断，从而可以减轻系统和 CPU 的负担。但由于我们讨论的 Linux 0.11 中仅使用了 NS8250/16450 的属性，因此这里不对 FIFO 方式作进一步说明。

PC 机中使用 UART 的异步串行口硬件逻辑见图 10-11 所示。其中可分成 3 部分。第一部分主要包括数据总线缓冲 D7 -- D0、内部寄存器选择引脚 A0 -- A2、CPU 读写数据选择通引脚 DISTR 和 DOSTR、芯片复位引脚 MR、中断请求输出引脚 INTRPT 以及用户自定义的用于禁止/允许中断的引脚 OUT2。当 OUT2 为 1 是可禁止 UART 发出中断请求信号。

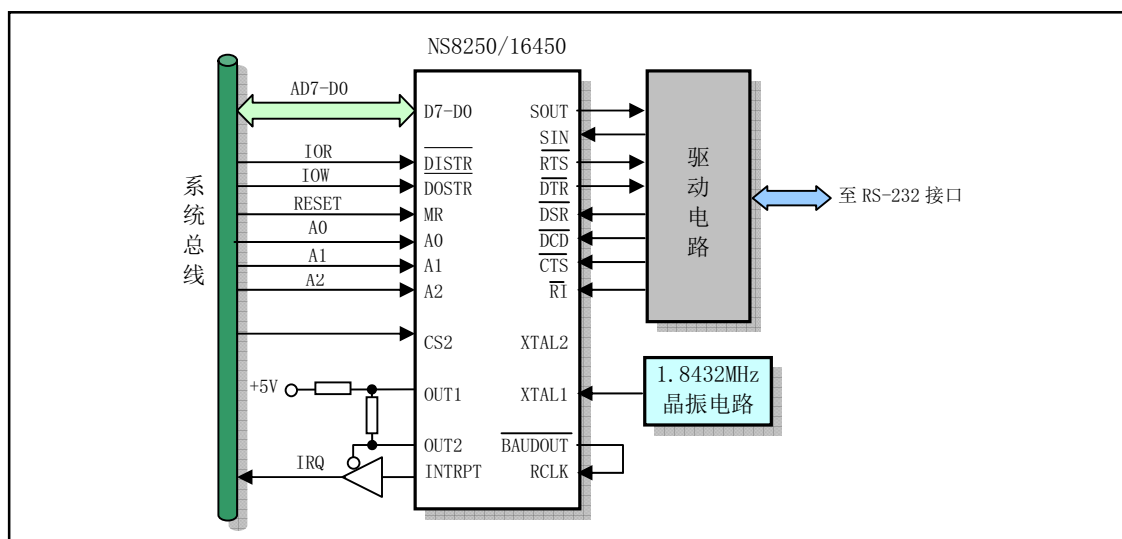


图 10-11 NS8250/16450 基本硬件配置结构图

第二部分主要包括 UART 与 RS-232 接口的引脚部分。这些引脚主要用于接收/发送串行数据和产生或接收 MODEM 控制信号。串行输出数据（SOUT）引脚向线路上发送比特数据流；输入数据（SIN）引脚接收线路上来的比特数据流；数据设备就绪（DSR）引脚用于通信设备（MODEM）通知 UART 准备好可以开始接收数据；数据终端就绪（DTR）引脚则用于计算机通知 MODEM 已准备好接收数据；请求发送（RTS）引脚用于通知 MODEM 计算机要求切换到发送方式；清除发送（CTS）则是 MODEM 告诉计算机已切换到准备接收方式；载波检测（DCD）引脚用于接收 MODEM 告知已接收到载波信号；振铃指示（RI）引脚也用于 MODEM 告诉计算机通信线路已经接通。

第三部分是 UART 芯片时钟输入电路部分。UART 的工作时钟可以通过在引脚 XTAL1、XTAL2 之间连接一个晶体振荡器来产生，也可以通过 XTAL1 直接从外部引入。PC 机则使用了后一种办法，在 XTAL1 引脚上直接输入 1.8432MHz 的时钟信号。UART 发送波特率的 16 倍由引脚 BAUDOUT 输出，而引脚 RCLK 是接收数据的波特率。由于这两者连接在一起，因此 PC 机上发送和接收数据波特率相同。

与中断控制芯片 8259A 一样，UART 也是一个可编程的控制芯片。通过对它其内部寄存器进行设置，我们可以设置串行通信的工作参数和 UART 的工作方式。UART 的内部组成框图见图 10-12 所示。

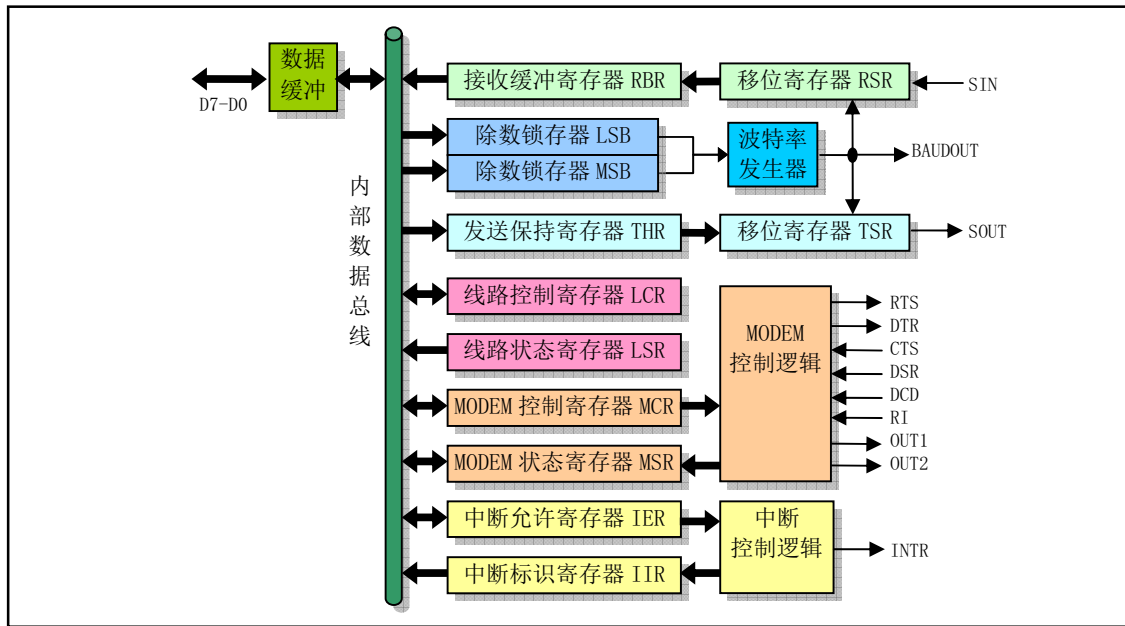


图 10-12 NS8250/16450 内部组成框图

NS8250 中 CPU 能够访问的寄存器有 10 个，但是用于选择这些寄存器的地址线 A2--A0 最多能够选择 8 个寄存器。因此 NS8250 中就在线路控制寄存器中拿出一位（位 7）用作选择两个除数锁存寄存器 LSB 和 MSB。位 7 就被称为除数锁存访问位 DLAB（Divisor Latch Access Bit）。这些寄存器的用途以及访问端口地址见表 10-8 所示。

表 10-8 UART 内部寄存器对应端口及用途

端口	读/写	条件	用途
0x3f8 (0x2f8)	写	DLAB=0	写发送保持寄存器 THR。含有将发送的字符。
	读	DLAB=0	读接收缓冲寄存器 RBR。含有收到的字符。
	读/写	DLAB=1	读/写波特率因子低字节（LSB）。
0x3f9 (0x2f9)	读/写	DLAB=1	读/写波特率因子高字节（MSB）。
	读/写	DLAB=0	读/写中断允许寄存器 IER。 位 7-4 全 0 保留不用； 位 3=1 modem 状态中断允许； 位 2=1 接收器线路状态中断允许； 位 1=1 发送保持寄存器空中断允许； 位 0=1 已接收到数据中断允许。
0x3fa (0x2fa)	读		读中断标识寄存器 IIR。中断处理程序用以判断此次中断是 4 种中的那一种。 位 7-3 全 0（不用）； 位 2-1 确定中断的优先级： = 11 接收状态有错中断，优先级最高。读线路状态可复位； = 10 已接收到数据中断，优先级 2。读接收数据可复位； = 01 发送保持寄存器空中断，优先级 3。写发送保持可复位； = 00 MODEM 状态改变中断，优先级 4。读 MODEM 状态可复位。

			位 0=0 有待处理中断；=1 无中断。
0x3fb (0x2fb)	写		<p>写线路控制寄存器 LCR。</p> <p>位 7=1 除数锁存访问位(DLAB)； =0 接收器，发送保持或中断允许寄存器访问；</p> <p>位 6=1 允许间断；</p> <p>位 5=1 保持奇偶位；</p> <p>位 4=1 偶校验；=0 奇校验；</p> <p>位 3=1 允许奇偶校验；=0 无奇偶校验；</p> <p>位 2=1 此时依赖于数据位长度。若数据位长度是 5 位，则停止位为 1.5 位；若数据位长度是 6、7 或 8 位，则停止位为 2 位； =0 停止位是 1 位；</p> <p>位 1-0 数据位长度： =00 5 位数据位； =01 6 位数据位； =10 7 位数据位； =11 8 位数据位。</p>
0x3fc (0x2fc)	写		<p>写 modem 控制寄存器 MCR。</p> <p>位 7-5 全 0 保留；</p> <p>位 4=1 芯片处于循环反馈诊断操作模式；</p> <p>位 3=1 辅助用户指定输出 2，允许 INTRPT 到系统；</p> <p>位 2=1 辅助用户指定输出 1，PC 机未用；</p> <p>位 1=1 使请求发送 RTS 有效；</p> <p>位 0=1 使数据终端就绪 DTR 有效。</p>
0x3fd (0x2fd)	读		<p>读线路状态寄存器 LSR。</p> <p>位 7=0 保留；</p> <p>位 6=1 发送移位寄存器为空；</p> <p>位 5=1 发送保持寄存器为空，可以取字符发送；</p> <p>位 4=1 接收到满足间断条件的位序列；</p> <p>位 3=1 帧格式错误；</p> <p>位 2=1 奇偶校验错误；</p> <p>位 1=1 超越覆盖错误；</p> <p>位 0=1 接收器数据准备好，系统可读取。</p>
0x3fe (0x2fe)	读		<p>读 MODEM 状态寄存器 MSR。δ 表示信号或条件发生变化。</p> <p>位 7=1 载波检测(CD)有效；</p> <p>位 6=1 响铃指示(RI)有效；</p> <p>位 5=1 数据设备就绪(DSR)有效；</p> <p>位 4=1 清除发送 (CTS) 有效；</p> <p>位 3=1 检测到 δ 载波；</p> <p>位 2=1 检测到响铃信号边沿；</p> <p>位 1=1 δ 数据设备就绪(DSR)；</p> <p>位 0=1 δ 清除发送(CTS)。</p>

2. UART 初始化编程方法

当 PC 机上电启动时，系统 RESET 信号通过 NS8250 的 MR 引脚使得 UART 内部寄存器和控制逻辑

复位。此后若要使用 UART 就需要对其进行初始化编程操作，以设置 UART 的工作波特率、数据位数以及工作方式等。下面我们以 PC 上的串行端口 1 为例说明对其初始化的步骤。该串口的端口基地址是 `port = 0x3f8`，UART 芯片中断引脚 `INTRPT` 被连接至中断控制芯片引脚 `IRQ4` 上。当然，在初始化之前应该首先在 IDT 表中设置好串行中断处理过程的中断描述符项。

a) 设置通信的传输波特率。

设置通信传输波特率就是设置两个除数锁存寄存器 `LSB` 和 `MSB` 的值，即 16 位的波特率因子。由上表可知，若要访问这两个除数锁存寄存器，我们必须首先设置线路控制寄存器 `LCR` 的第 8 位 `DLAB=1`，即向端口 `port+3 (0x3fb)` 写入 `0x80`。然后对端口 `port (0x3f8)` 和 `port+1 (0x3f9)` 执行输出操作即可把波特率因子分别写入 `LSB` 和 `MSB` 中。对于指定的波特率（例如 2400bps），波特率因子的计算公式为：

$$\text{波特率因子} = \frac{\text{UART时钟频率}}{\text{波特率} \times 16} = \frac{1.8432\text{MHz}}{2400 \times 16} = \frac{1843200}{2400 \times 16} = 48$$

因此若要设置波特率为 2400pbs，我们需要在 `LSB` 中写入 `0x30`，在 `MSB` 中写入 `0`。波特率设置好后，我们最好还需要复位线路控制寄存器的 `DLAB` 位。

b) 设置通信传输格式。

串行通信传输格式由线路控制寄存器 `LCR` 中的各位来定义。其中每位的含义见上表所示。如果我们需要把传输格式设置成无奇偶校验位、8 位数据位和 1 位停止位，那么就需要向 `LCR` 输出值 `0x03`。`LCR` 最低 2 位表示数据位长度，当为 11 时表示数据长度是 8 位。

c) 设置 MODEM 控制寄存器。

对该寄存器进行写入操作可以设置 UART 的操作方式和控制 MODEM。UART 操作方式有中断方式和查询方式两种。还有一种循环反馈方式，但该方式仅用于诊断测试 UART 芯片的好坏，不能作为一种实际的通信方式使用。在 PC 机 ROM BIOS 中使用的是查询方式，但本书讨论的 Linux 系统采用的是高效率的中断方式。因此我们将在下面只介绍中断方式下 UART 的操作编程方法。

设置 `MCR` 的位 4 可让 UART 处于循环反馈诊断操作方式下。在这种方式下 UART 芯片内部自动把输入（`SIN`）和输出（`SOUT`）引脚“短接”，因此若此时发送的数据序列和接收到的序列相等，那么就说明 UART 芯片工作正常。

中断方式是指当 MODEM 状态发生变化时、或者接收出错时、或者发送保持寄存器空时、或者接收到一个字符时允许 UART 通过 `INTRPT` 引脚向 CPU 发出中断请求信号。至于允许那些条件下发出中断请求则由中断允许寄存器 `IER` 来确定。但是若要让 UART 的中断请求信号能够送到 8259A 中断控制器去，就需要把 MODEM 控制寄存器 `MCR` 的位 3（`OUT2`）置位。因为在 PC 机中，该位控制着 `INTRPT` 引脚到 8259A 的电路，参见图 10-11 所示。

查询方式是指 MODEM 控制寄存器 `MCR` 位 3（`OUT2`）复位的条件下，程序通过循环查询 UART 寄存器的内容来接收/发送串行数据。当 `MCR` 的位 3=0 时，虽然在 MODEM 状态发生变化等条件下 UART 仍然能在 `INTRPT` 引脚产生中断请求信号，并且能根据产生中断的条件设置中断标识寄存器 `IIR` 的内容，但是中断请求信号并不能被送到 8259A 中。因此程序只能通过查询线路状态寄存器 `LSR` 和中断标识寄存器 `IIR` 的内容来判断 UART 的当前工作状态并进行数据的接收和发送操作。

`MCR` 的位 1 和位 0 分别用于控制 MODEM，当这两位置位时，UART 的数据终端就绪 `DTR` 引脚和请求发送 `RTS` 引脚输出有效。

若要把 UART 设置成中断方式，并且使 `DTR` 和 `RTS` 有效，那么我们就需要向 MODEM 控制寄存器写入 `0x0b`，即二进制数 01011。

d) 初始化中断允许寄存器。

中断允许寄存器 IER 用来设置可产生中断的条件，即中断来源类型。共有 4 种中断源类型可供选择，见表 10-8 所示。对应比特位置 1 表示允许该条件产生中断，否则禁止。当某个中断源类型产生了中断，那么具体是哪个中断源产生的中断就有中断标识寄存器 IIR 中的位 2--位 1 指明，并且读写特定寄存器的内容可以复位 UART 的中断。IER 的位 0 用于确定当前是否有中断，位 0=0 表示有待处理的中断。

在 Linux 0.11 串行端口初始化函数中，设置允许 3 种中断源产生中断(写入 0x0d)，即在 MODEM 状态发生变化时、在接收有错时、在接收器收到字符时都允许产生中断，但不允许发送保持寄存器空产生中断。因为我们此时还没有数据要发送。当对应串行终端的写队列有数据要发送出去时，tty_write()函数会调用 rs_write()函数来置位发送保持寄存器空允许中断标志，从而在该中断源引发的串行中断处理过程中内核程序就可以开始取出写队列中的字符发送输出到发送保持寄存器中，让 UART 发送出去。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。

3. UART 中断处理程序编程方法

Linux 内核中，串行终端使用读/写队列来接收和发送终端数据。从串行端口接收到的数据被放入读队列头指针处，供 tty_io.c 程序来读取；需要发送到串行终端去的数据被放到了写队列头指针处。因此串行中断处理程序的主要任务就是把 UART 接收到的接收缓冲寄存器 RBR 中的字符放到读队列尾指针处；从写队列尾指针处取出字符放进 UART 的发送保持寄存器 THR 中发送出去。同时串行中断处理程序还需要处理其他一些出错情况。

由上面说明可知，UART 可有 4 种不同的中断源类型产生中断。因此当串行中断处理程序刚开始执行时仅知道发生了中断，但不知道是哪个情况引起了中断。所以串行中断处理程序的第一个任务就是确定产生中断的具体条件。这需要借助于中断标识寄存器 IIR 来确定产生当前中断的源类型。因此串行中断处理程序可以根据产生中断的源类型使用子程序地址跳转表 jmp_table[]来分别处理，其框图见图 10-13 所示。rs_io.s 程序的结构与这个框图基本相同。

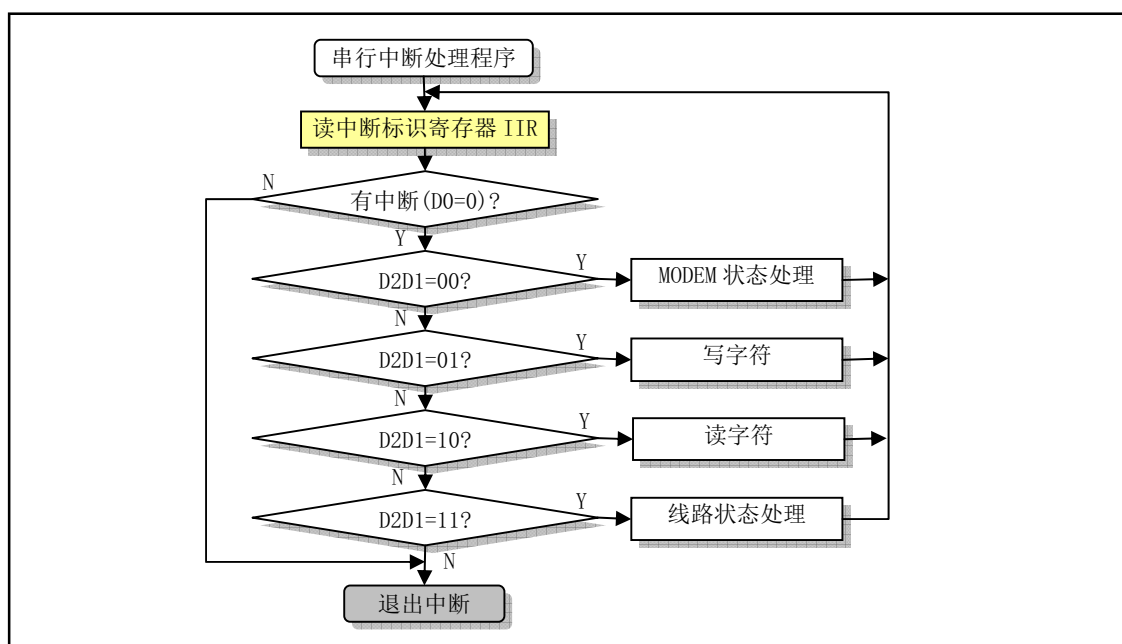


图 10-13 串行通信中断处理程序框图

在取出 IIR 的内容后，需要首先根据位 0 判断是否有待处理的中断。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。

10.6 rs_io.s 程序

10.6.1 功能描述

该汇编程序实现 rs232 串行通信中断处理过程。在进行字符的传输和存储过程中，该中断过程主要对终端的读、写缓冲队列进行操作。它把从串行线路上接收到的字符存入串行终端的读缓冲队列 read_q 中，或把写缓冲队列 write_q 中需要发送出去的字符通过串行线路发送给远端的串行终端设备。

引起系统发生串行中断的情况有 4 种：a. 由于 modem 状态发生了变化；b. 由于线路状态发生了变化；c. 由于接收到字符；d. 由于在中断允许标志寄存器中设置了发送保持寄存器中断允许标志，需要发送字符。对引起中断的前两种情况的处理过程是通过读取对应状态寄存器值，从而使其复位。对于由于接收到字符的情况，程序首先把该字符放入读缓冲队列 read_q 中，然后调用 copy_to_cooked() 函数转换成以字符行为单位的规范模式字符放入辅助队列 secondary 中。对于需要发送字符的情况，则程序首先从写缓冲队列 write_q 尾指针处取出一个字符发送出去，再判断写缓冲队列是否已空，若还有字符则循环执行发送操作。

因此，在阅读本程序之前，最好先看一下 include/linux/tty.h 头文件。其中给出了字符缓冲队列的数据结构 tty_queue、终端的数据结构 tty_struct 和一些控制字符的值。另外还有一些对缓冲队列进行操作的宏定义。缓冲队列及其操作示意图请参见图 10-14 所示。

10.6.2 代码注释

程序 10-5 linux/kernel/chr_drv/rs_io.s

```

1 /*
2  * linux/kernel/rs_io.s
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  *      rs_io.s
9  *
10 * This module implements the rs232 io interrupts.
11 */
12 /*
13  * 该模块实现 rs232 输入输出中断处理程序。
14 */
15
16 .text
17 .globl _rs1_interrupt, _rs2_interrupt

```

```

15 // 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。
16 size      = 1024                                /* must be power of two !
17                                                    and must match the value
18                                                    in tty_io.c!!! */
19
20 /* these are the offsets into the read/write buffer structures */
21 /* 以下这些是读写缓冲队列结构中的偏移量 */
22 // 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
23 // 对应 tty_queue 结构的数据字段。对于串行终端缓冲队列，该字段存放着串行端口基地址。
21 rs_addr = 0                                     // 串行端口号字段偏移（端口是 0x3f8 或 0x2f8）。
22 head = 4                                       // 缓冲区中头指针字段偏移。
23 tail = 8                                       // 缓冲区中尾指针字段偏移。
24 proc_list = 12                                // 等待该缓冲的进程字段偏移。
25 buf = 16                                       // 缓冲区字段偏移。
26
27 // 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
28 // 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。
27 startup = 256                                /* chars left in write queue when we restart it */
28                                                /* 当我们重新开始写时，队列里最多还剩余字符个数。*/
29
30 /*
31 * These are the actual interrupt routines. They look where
32 * the interrupt is coming from, and take appropriate action.
33 */
34 /*
35 * 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
36 * 相应的处理。
37 */
33 .align 2
34 // 串行端口 1 中断处理程序入口点。
35 // 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。
36 // 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，99），
37 // 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
38 // 结构格式请参见 include/linux/tty.h，第 16 行。
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
38 // 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。
40
41 // 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
42 // 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
43 // 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
44 // 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
45 // 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。
46 // 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
47 // 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
48 // 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 */ /* 不乘 4，位 0 已是 0 */
// 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      shr $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。指针不能超过缓冲区大小。
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:   pushl %edx             // 将串口号压入栈 (1- 串口 1, 2 - 串口 2), 作为参数。
109     call _do_tty_interrupt // 调用 tty 中断处理 C 函数 (tty_io.c, 342 行)。
110     addl $4,%esp            // 丢弃入栈参数, 并返回。
111     ret
112
// 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。
113 .align 2
114 write_char:
115     movl 4(%ecx),%ecx        # write-queue // 取写缓冲队列结构地址 → ecx。
116     movl head(%ecx),%ebx     // 取写队列头指针 → ebx。
117     subl tail(%ecx),%ebx     // 头指针 - 尾指针 = 队列中字符数。
118     andl $size-1,%ebx       # nr chars in queue # 对指针取模运算。
119     je write_buffer_empty    // 若头指针 = 尾指针, 说明写队列空, 跳转处理。
120     cmpl $startup,%ebx       // 队列中字符数还超过 256 个?
121     ja 1f                   // 超过则跳转处理。
122     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。
123     testl %ebx,%ebx         # is there any? # 有等待写的进程吗?
124     je 1f                   // 是空的, 则向前跳转到标号 1 处。
125     movl $0,(%ebx)          // 否则将进程置为可运行状态 (唤醒进程)。

```



```

126 1:      movl tail(%ecx),%ebx      // 取尾指针。
127      movb buf(%ecx,%ebx),%al    // 从缓冲中尾指针处取一字符→al。
128      outb %al,%dx              // 向端口 0x3f8 (0x2f8) 写到发送保持寄存器中。
129      incl %ebx                 // 尾指针前移。
130      andl $size-1,%ebx          // 尾指针若到缓冲区末端，则折回。
131      movl %ebx,tail(%ecx)        // 保存已修改过的尾指针。
132      cmpl head(%ecx),%ebx        // 尾指针与头指针比较，
133      je write_buffer_empty      // 若相等，表示队列已空，则跳转。
134      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 就会“自动”地来取写缓冲队列中的字符，并发送出去。
135 .align 2
136 write_buffer_empty:
137      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。
138      testl %ebx,%ebx            # is there any? # 有等待的进程吗？
139      je 1f                      // 无，则向前跳转到标号 1 处。
140      movl $0, (%ebx)            // 否则将进程置为可运行状态（唤醒进程）。
141 1:      incl %edx                // 指向端口 0x3f9 (0x2f9)。
142      inb %dx,%al                // 读取中断允许寄存器 IER。
143      jmp 1f                      // 稍作延迟。
144 1:      jmp 1f                  /* 屏蔽发送保持寄存器空中断（位 1） */
145 1:      andb $0xd,%al           /* disable transmit interrupt */
146      outb %al,%dx              // 写入 0x3f9 (0x2f9)。
147      ret

```

10.7 tty_io.c 程序

10.7.1 功能描述

每个 tty 设备有 3 个缓冲队列，分别是读缓冲队列（read_q）、写缓冲队列（write_q）和辅助缓冲队列（secondary），定义在 tty_struct 结构中（include/linux/tty.h）。对于每个缓冲队列，读操作是从缓冲队列的左端取字符，并且把缓冲队列尾（tail）指针向右移动。而写操作则是往缓冲队列的右端添加字符，并且也把头(head)指针向右移动。这两个指针中，任何一个若移动到超出了缓冲队列的末端，则折回到左端重新开始。见图 10-14 所示。

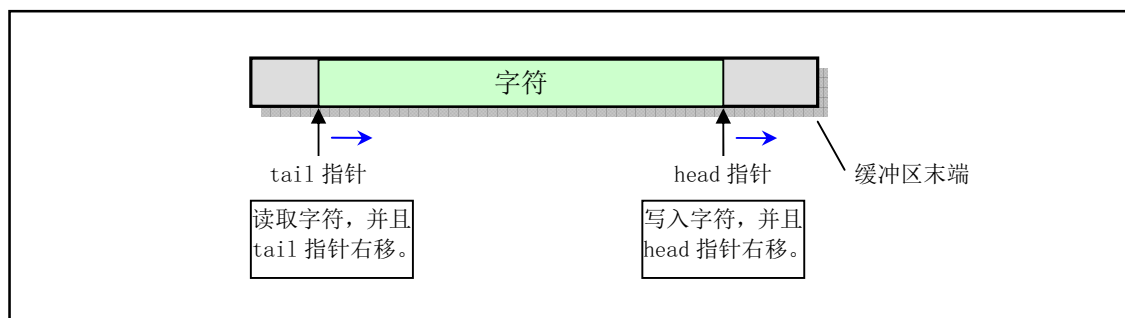


图 10-14 tty 字符缓冲队列的操作方式

本程序包括字符设备的上层接口函数。主要含有终端读/写函数 `tty_read()` 和 `tty_write()`。读操作的行规则函数 `copy_to_cooked()` 也在这里实现。

`tty_read()` 和 `tty_write()` 将在文件系统中用于操作字符设备文件时被调用。例如当一个程序读 `/dev/tty` 文件时, 就会执行系统调用 `sys_read()` (在 `fs/read_write.c` 中), 而这个系统调用在判别出所读文件是一个字符设备文件时, 即会调用 `rw_char()` 函数 (在 `fs/char_dev.c` 中), 该函数则会根据所读设备的子设备号等信息, 由字符设备读写函数表 (设备开关表) 调用 `rw_tty()`, 最终调用到这里的终端读操作函数 `tty_read()`。

`copy_to_cooked()` 函数由键盘中断过程调用 (通过 `do_tty_interrupt()`), 用于根据终端 `termios` 结构中设置的字符输入/输出标志 (例如 `INLCR`、`OUCLC`) 对 `read_q` 队列中的字符进行处理, 把字符转换成以字符行为单位的规范模式字符序列, 并保存在辅助字符缓冲队列 (规范模式缓冲队列) (`secondary`) 中, 供上述 `tty_read()` 读取。在转换处理期间, 若终端的回显标志 `L_ECHO` 置位, 则还会把字符放入写队列 `write_q` 中, 并调用终端写函数把该字符显示在屏幕上。如果是串行终端, 那么写函数将是 `rs_write()` (在 `serial.c`, 53 行)。`rs_write()` 会把串行终端写队列中的字符通过串行线路发送给串行终端, 并显示在串行终端的屏幕上。`copy_to_cooked()` 函数最后还将唤醒等待着辅助缓冲队列的进程。函数实现的步骤如下所示:

1. 如果读队列空或者辅助队列已经满, 则跳转到最后一步 (第 10 步), 否则执行以下操作;
2. 从读队列 `read_q` 的尾指针处取一字符, 并且尾指针前移一字符位置;
3. 若是回车 (CR) 或换行 (NL) 字符, 则根据终端 `termios` 结构中输入标志 (`ICRNL`、`INLCR`、`INOCR`) 的状态, 对该字符作相应转换。例如, 如果读取的是一个回车字符并且 `ICRNL` 标志是置位的, 则把它替换成换行字符;
4. 若大写转小写标志 `IUCLC` 是置位的, 则把字符替换成对应的小写字符;
5. 若规范模式标志 `ICANON` 是置位的, 则对该字符进行规范模式处理:
 - a. 若是删行字符 (^U), 则删除 `secondary` 中的一行字符 (队列头指针后退, 直到遇到回车或换行或队列已空为止);
 - b. 若是擦除字符 (^H), 则删除 `secondary` 中头指针处的一个字符, 头指针后退一个字符位置;
 - c. 若是停止字符 (^S), 则设置终端的停止标志 `stopped=1`;
 - d. 若是开始字符 (^Q), 则复位终端的停止标志。
6. 如果接收键盘信号标志 `ISIG` 是置位的, 则为进程生成对应键入控制字符的信号;
7. 如果是行结束字符 (例如 NL 或 ^D), 则辅助队列 `secondary` 的行数统计值 `data` 增 1;
8. 如果本地回显标志是置位的, 则把字符也放入写队列 `write_q` 中, 并调用终端写函数在屏幕上显示该字符;
9. 把该字符放入辅助队列 `secondary` 中, 返回上面第 1 步继续循环处理读队列中其他字符;
10. 最后唤醒睡眠在辅助队列上的进程。

在阅读下面程序时不免首先查看一下 `include/linux/tty.h` 头文件。在该头文件定义了 `tty` 字符缓冲队列

的数据结构以及一些宏操作定义。另外还定义了控制字符的 ASCII 码值。

10.7.2 代码注释

程序 10-6 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.
12  */
13 /*
14  * '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。
15  * 该程序同样实现了回显、规范(熟)模式等。
16  *
17  * Kill-line, 谢谢 John T Kohl.
18  */
19 #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。
20 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。
21 #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。
22
23 // 下面给出一些信号在信号位图中对应的比特屏蔽位。
24 #define ALRMASK (1<<(SIGALRM-1)) // 警告(alarm)信号屏蔽位。
25 #define KILLMASK (1<<(SIGKILL-1)) // 终止(kill)信号屏蔽位。
26 #define INTMASK (1<<(SIGINT-1)) // 键盘中断(int)信号屏蔽位。
27 #define QUITMASK (1<<(SIGQUIT-1)) // 键盘退出(quit)信号屏蔽位。
28 #define TSTPMASK (1<<(SIGTSTP-1)) // tty 发出的停止进程(tty stop)信号屏蔽位。
29
30 #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。
31 #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。
32 #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。
33 #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。
34
35 // 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。
36 #define L_FLAG(tty, f) ((tty->termios.c_lflag & f) // 本地模式标志。
37 #define I_FLAG(tty, f) ((tty->termios.c_iflag & f) // 输入模式标志。
38 #define O_FLAG(tty, f) ((tty->termios.c_oflag & f) // 输出模式标志。
39
40 // 取 termios 结构终端特殊(本地)模式标志集中的一个标志。
41 #define L_CANON(tty) L_FLAG((tty), ICANON) // 取规范模式标志。
42 #define L_ISIG(tty) L_FLAG((tty), ISIG) // 取信号标志。
43 #define L_ECHO(tty) L_FLAG((tty), ECHO) // 取回显字符标志。
44 #define L_ECHOE(tty) L_FLAG((tty), ECHOE) // 规范模式时取回显擦出标志。
45 #define L_ECHOK(tty) L_FLAG((tty), ECHOK) // 规范模式时取 KILL 擦除当前行标志。
46 #define L_ECHOCTL(tty) L_FLAG((tty), ECHOCTL) // 取回显控制字符标志。
47 #define L_ECHOKL(tty) L_FLAG((tty), ECHOKL) // 规范模式时取 KILL 擦除行并回显标志。

```

```

39 // 取 termios 结构输入模式标志集中的一个标志。
40 #define I_UCLC(tty)      I_FLAG((tty), I_UCLC) // 取大写到低写转换标志。
41 #define I_NLCR(tty)      I_FLAG((tty), I_NLCR) // 取换行符 NL 转回车符 CR 标志。
42 #define I_CRNL(tty)      I_FLAG((tty), I_CRNL) // 取回车符 CR 转换行符 NL 标志。
43 #define I_NOCR(tty)      I_FLAG((tty), I_NOCR) // 取忽略回车符 CR 标志。
44
45 // 取 termios 结构输出模式标志集中的一个标志。
46 #define O_POST(tty)      O_FLAG((tty), O_POST) // 取执行输出处理标志。
47 #define O_NLCR(tty)      O_FLAG((tty), O_NLCR) // 取换行符 NL 转回车换行符 CR-NL 标志。
48 #define O_CRNL(tty)      O_FLAG((tty), O_CRNL) // 取回车符 CR 转换行符 NL 标志。
49 #define O_NLRET(tty)      O_FLAG((tty), O_NLRET) // 取换行符 NL 执行回车功能的标志。
50 #define O_LCUC(tty)      O_FLAG((tty), O_LCUC) // 取小写转大写字母标志。
51
52 // tty 数据结构的 tty_table 数组。其中包含三个初始化项数据，分别是控制台、串口终端 1 和
53 // 串口终端 2 的初始化数据。
54 struct tty_struct tty_table[] = {
55     {
56         { I_CRNL, /* change incoming CR to NL */ /*将输入 CR 转换为 NL*/
57         O_POST | O_NLCR, /* change outgoing NL to CRNL */ /*将输出 NL 转 CRNL*/
58         0, // 控制模式标志集。
59         ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // 本地模式标志。
60         0, /* console termio */ // 线路规程，0 - TTY。
61         INIT_C CC, // 控制字符数组。
62         0, /* initial pgrp */ // 所属初始进程组。
63         0, /* initial stopped */ // 初始停止标志。
64         con write, // 控制台写函数。
65         {0, 0, 0, 0, ""}, /* console read-queue */ // 控制台读缓冲队列。
66         {0, 0, 0, 0, ""}, /* console write-queue */ // 控制台写缓冲队列。
67         {0, 0, 0, 0, ""} /* console secondary queue */ // 控制台辅助(第 2)队列。
68     }, {
69         {0, /* no translation */ // 输入模式标志集。0，无须转换。
70         0, /* no translation */ // 输出模式标志集。0，无须转换。
71         B2400 | CS8, // 控制模式标志集。2400bps，8 位数据位。
72         0, // 本地模式标志集。
73         0, // 线路规程，0 - TTY。
74         INIT_C CC, // 控制字符数组。
75         0, // 所属初始进程组。
76         0, // 初始停止标志。
77         rs write, // 串口 1 终端写函数。
78         {0x3f8, 0, 0, 0, ""}, /* rs 1 */ // 串行终端 1 读缓冲队列结构初始值。
79         {0x3f8, 0, 0, 0, ""}, // 串行终端 1 写缓冲队列结构初始值。
80         {0, 0, 0, 0, ""} // 串行终端 1 辅助缓冲队列结构初始值。
81     }, {
82         {0, /* no translation */ // 输入模式标志集。0，无须转换。
83         0, /* no translation */ // 输出模式标志集。0，无须转换。
84         B2400 | CS8, // 控制模式标志集。2400bps，8 位数据位。
85         0, // 本地模式标志集。
86         0, // 线路规程，0 - TTY。
87         INIT_C CC, // 控制字符数组。
88         0, // 所属初始进程组。
89         0, // 初始停止标志。
90         rs write, // 串口 2 终端写函数。

```

```

88         {0x2f8, 0, 0, 0, ""}, /* rs 2 */ // 串行终端 2 读缓冲队列结构初始值。
89         {0x2f8, 0, 0, 0, ""}, // 串行终端 2 写缓冲队列结构初始值。
90         {0, 0, 0, 0, ""} // 串行终端 2 辅助缓冲队列结构初始值。
91     }
92 };
93
94 /*
95  * these are the tables used by the machine code handlers.
96  * you can implement pseudo-tty's or something by changing
97  * them. Currently not done.
98  */
99 /*
100  * 下面是汇编程序使用的缓冲队列结构地址表。通过修改这个表，
101  * 你可以实现伪 tty 终端或其他终端类型。目前还没有这么做。
102  */
103 // tty 读写缓冲队列结构地址表。供 rs_io.s 汇编程序使用，用于取得读写缓冲队列地址。
104 struct tty_queue * table_list[]={
105     &tty_table[0].read_q, &tty_table[0].write_q, // 控制台终端读写队列地址。
106     &tty_table[1].read_q, &tty_table[1].write_q, // 串行终端 1 读写队列地址。
107     &tty_table[2].read_q, &tty_table[2].write_q // 串行终端 2 读写队列地址。
108 };
109
110 //tty 终端初始化函数。
111 // 初始化串口终端和控制台终端。
112 void tty_init(void)
113 {
114     rs_init(); // 初始化串行中断程序和串行接口 1 和 2 (serial.c, 37 行)。
115     con_init(); // 初始化控制台终端 (console.c, 617 行)。
116 }
117
118 //tty 键盘中断字符 (^C) 处理函数。
119 // 向 tty 结构中指定的 (前台) 进程组中所有进程发送指定信号 mask，通常该信号是 SIGINT。
120 // 参数: tty - 指定终端的 tty 结构指针; mask - 信号屏蔽位。
121 void tty_intr(struct tty_struct * tty, int mask)
122 {
123     int i;
124
125     // 首先检查终端进程组号。如果 tty 所属进程组号小于等于 0，则退出。因为当 pgrp = 0 时，
126     // 表明进程是初始进程 init，它没有控制终端，因此不应该会发出中断字符。
127     if (tty->pgrp <= 0)
128         return;
129     // 扫描任务数组，向 tty 指定的进程组 (前台进程组) 中所有进程发送指定信号。即如果该项
130     // 任务指针不为空，并且其组号等于 tty 组号，则设置 (发送) 该任务指定的信号 mask。
131     for (i=0; i<NR_TASKS; i++)
132         if (task[i] && task[i]->pgrp==tty->pgrp)
133             task[i]->signal |= mask;
134 }
135
136 //tty 如果队列缓冲区空则让进程进入可中断睡眠状态。
137 // 参数: queue - 指定队列的指针。
138 // 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。
139 static void sleep_if_empty(struct tty_queue * queue)
140 {

```

```

// 若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。
124     cli(); // 关中断。
125     while (!current->signal && EMPTY(*queue))
126         interruptible_sleep_on(&queue->proc_list);
127     sti(); // 开中断。
128 }
129
//// 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。
130 static void sleep_if_full(struct tty_queue * queue)
131 {
// 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。
132     if (!FULL(*queue))
133         return;
134     cli();
135     while (!current->signal && LEFT(*queue)<128)
136         interruptible_sleep_on(&queue->proc_list);
137     sti();
138 }
139
//// 等待按键。
// 如果控制台读队列缓冲区空，则让进程进入可中断睡眠状态。
140 void wait_for_keypress(void)
141 {
142     sleep_if_empty(&tty_table[0].secondary);
143 }
144
//// 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。
145 void copy_to_cooked(struct tty_struct * tty)
146 {
147     signed char c;
148
// 如果 tty 的读队列缓冲区不空并且辅助队列缓冲区不满，则循环读取读队列缓冲区中的字符，
// 转换成规范模式后放入辅助队列（secondary）缓冲区中，直到读队列缓冲区空或者辅助队
// 列满为止。在循环体内，程序首先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。然后根据终端 termios 中输入模式标志集中设置的标志对字符进行处理。
149     while (!EMPTY(tty->read_q) && !FULL(tty->secondary)) {
150         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）。
151         if (c==13)
152             if (I_CRNL(tty))
153                 c=10;
154             else if (I_NOCR(tty))
155                 continue;
156             else ; // 这个'else'用于结束内部'if'语句。

```

```

157         else if (c==10 && I_NLCR(tty))
158             c=13;
// 如果大写转小写标志 UCLC 置位，则将该字符转换为小写字符。
159         if (I_UCLC(tty))
160             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
// 符的循环过程如下：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。
161         if (L_CANON(tty)) {
162             if (c==KILL_CHAR(tty)) {
163                 /* deal with killing the input line */ /* 删除输入行*/
164                 while(!(EMPTY(tty->secondary) ||
165                     (c=LAST(tty->secondary))==10 ||
166                     c==EOF_CHAR(tty))) {
167                     if (L_ECHO(tty)) { // 若本地回显标志置位。
168                         if (c<32) // 控制字符要删 2 字节。
169                             PUTCH(127, tty->write_q);
170                             PUTCH(127, tty->write_q);
171                             tty->write(tty);
172                     }
173                     DEC(tty->secondary. head);
174                 }
175                 continue; // 继续读取读队列中字符进行处理。
176             }
// 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。
177             if (c==ERASE_CHAR(tty)) {
178                 if (EMPTY(tty->secondary) ||
179                     (c=LAST(tty->secondary))==10 ||
180                     c==EOF_CHAR(tty))
181                     continue;
182                 if (L_ECHO(tty)) { // 若本地回显标志置位。
183                     if (c<32)
184                         PUTCH(127, tty->write_q);
185                         PUTCH(127, tty->write_q);
186                         tty->write(tty);
187                 }
188                 DEC(tty->secondary. head);
189                 continue;
190             }
// 如果字符是停止控制字符 (^S)，则置 tty 停止标志，停止 tty 输出，并继续处理其他字符。
// 如果字符是开始字符 (^Q)，则复位 tty 停止标志，恢复 tty 输出，并继续处理其他字符。
191             if (c==STOP_CHAR(tty)) {
192                 tty->stopped=1;

```



```

193         continue;
194     }
195     if (c==START_CHAR(tty)) {
196         tty->stopped=0;
197         continue;
198     }
199 } // 当前字符的输入规范模式处理结束 (L_CANON(tty))。

// 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，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，并继续处理下一字符。
200     if (L_ISIG(tty)) {
201         if (c==INTR_CHAR(tty)) { // 若是^C，则发送中断信号。
202             tty_intr(tty, INTMASK);
203             continue;
204         }
205         if (c==QUIT_CHAR(tty)) { // 若是^，则发送退出信号。
206             tty_intr(tty, QUITMASK);
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 264 行。
210     if (c==10 || c==EOF_CHAR(tty))
211         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 写队列缓冲池中；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。
212     if (L_ECHO(tty)) {
213         if (c==10) {
214             PUTCH(10, tty->write_q);
215             PUTCH(13, tty->write_q);
216         } else if (c<32) {
217             if (L_ECHOCTL(tty)) {
218                 PUTCH('^', tty->write_q);
219                 PUTCH(c+64, tty->write_q);
220             }
221         } else
222             PUTCH(c, tty->write_q);
223         tty->write(tty);
224     }

// 每一次循环末将处理过的字符放入辅助队列中。
225     PUTCH(c, tty->secondary);
226 }

// 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)。
227     wake_up(&tty->secondary.proc_list);
228 }
229

//// 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。
230 int tty_read(unsigned channel, char * buf, int nr)
231 {
232     struct tty_struct * tty;
233     char c, * b=buf;
234     int minimum, time, flag=0;
235     long oldalarm;
236
// 首先判断函数参数有效性, 并让 tty 指针指向参数子设备号对应 ttb_table 表中的 tty 结构。
// 本版本 Linux 内核终端只有 3 个子设备, 分别是控制台终端 (0)、串口终端 1 (1) 和串口终
// 端 2 (2)。所以任何大于 2 的子设备号都是非法的。需要读取的字节数当然也不能小于 0。
237     if (channel>2 || nr<0) return -1;
238     tty = &tty_table[channel];
// 接着保存进程原定时值, 并根据 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。
239     oldalarm = current->alarm; // 保存进程当前报警定时值 (滴答数)。
240     time = 10L*tty->termios.c_cc[VTIME]; // 设置读操作超时定时值。
241     minimum = tty->termios.c_cc[VMIN]; // 最少需要读取的字符个数。
// 然后根据 time 和 minimum 的数值设置最少需要读取的确切字符数和等待延时值。 如果设置了
// 读超时定时值 time (即不等于 0), 但没有设置最少读取个数 minimum, 即目前是 0, 那么我
// 们将设置成在读到至少一个字符或者定时超后读操作将立刻返回。所以需要置 minimum=1。
// 如果进程原定时值是 0 或者 time 加上当前系统时间值小于进程原定时值的话, 则置重新设置
// 进程定时值为 (time + 当前系统时间), 并置 flag 标志。另外, 如果这里设置的最少读取字
// 符数大于欲读取的字符数, 则令其等于此次欲读取的字符数 nr。
// 注意, 这里设置进程 alarm 值而导致收到的 SIGALRM 信号并不会导致进程终止退出。当 245 行
// 设置进程的 alarm 时, 同时也会设置一个 flag 标志。当这个 alarm 超时时, 虽然内核也会向
// 进程发送 SIGALRM 信号, 但这里会利用 flag 标志来判断到底是用户还是内核设置的 alarm 值,
// 若 flag 不为 0, 那么内核代码就会负责复位此时产生的 SIGALRM 信号 (第 251 行)。因此
// 这里设置的 SIGALRM 信号不会使当前进程“无故”终止退出。当然, 这种方式有些烦琐, 并且
// 容易出问题, 因此以后内核数据结构中就特地使用了一个 timeout 变量来专门处理这种问题。
242     if (time && !minimum) {
243         minimum=1;
244         if (flag=(!oldalarm || time+jiffies<oldalarm))
245             current->alarm = time+jiffies;
246     }
247     if (minimum>nr)
248         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 flag 已设置 (即进程原定时值是 0 或者 time +
// 当前系统时间值小于进程原定时值), 并且进程此时已收到定时报警信号 SIGALRM, 表明这
// 里新设置的定时时间已到。于是复位进程的定时信号 SIGALRM, 并中断循环。
// 如果 flag 没有置位 (即进程原来设置过定时值并且这里重新设置的定时值大于等于原来设置
// 的定时值) 因而是收到了原定时的报警信号, 或者 flag 已置位但当前进程此时收到了其他信
// 号, 则退出循环, 返回 0。
249     while (nr>0) {
250         if (flag && (current->signal & ALRMMASK)) {
251             current->signal &= ~ALRMMASK;
252             break;
253         }
254         if (current->signal) // 若是进程原定时到或者收到其他信号。
255             break;

```

```

// 如果辅助缓冲队列为空，或者设置了规范模式标志并且辅助队列中字符行数为 0 以及辅助模
// 式缓冲队列空闲空间>20，则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。另外，由这里的 LEFT() 判断可知，即使辅助队
// 列中还没有放入一行（即应该有一个回车符），但是如果此时一行字符个数已经超过 1024
// - 20 = 1004 个字符，那么内核也会立刻执行读取操作。
256         if (EMPTY(tty->secondary) || (L_CANON(tty) &&
257             !tty->secondary.data && LEFT(tty->secondary)>20)) {
258             sleep_if_empty(&tty->secondary);
259             continue;
260         }
// 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，则返
// 回已读字符数，并退出。
261         do {
262             GETCH(tty->secondary, c);
263             if (c==EOF_CHAR(tty) || c==10)
264                 tty->secondary.data--; // 字符行数减 1。
265             if (c==EOF_CHAR(tty) && L_CANON(tty))
266                 return (b-buf);
// 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。
// 否则，只要还没有取完欲读字符数并且辅助队列不空，就继续取队列中的字符。
// 另外，由于在规范模式下遇到换行符 NL 时我们也应该退出。所以在第 271 行后应该再添上一
// 条语句：“if (c==10 && L_CANON(tty)) break;”。
267             else {
268                 put_fs_byte(c, b++);
269                 if (!--nr)
270                     break;
271             }
272         } while (nr>0 && !EMPTY(tty->secondary));
// 执行到此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。对于已经读取 nr 个
// 字符的情况，我们无需做什么，程序会自动退出 249 行开始的循环，恢复进程原定时值并作
// 退出处理。如果由于辅助队列中字符被取空而退出上面 do 循环，那么我们就需要根据终端
// termios 控制字符数组的 time 设置的延时时间来确定是否要等待一段时间。如果超时时值
// time 不为 0 并且规范模式标志没有置位（非规范模式），那么：
// 如果进程原定时值是 0 或者 time+ 当前系统时间值 小于进程原定时值 oldalarm，则置重新
// 设置进程定时值为 time+当前系统时间，并置 flag 标志，为还需要读取的字符做好定时准备。
// 否则表明进程原定时时间要比等待字符被读取的定时时间要早，可能没有等到字符到来进程
// 原定时时间就到了。因此，此时这里需要恢复进程的原定时值 oldalarm。
273         if (time && !L_CANON(tty))
274             if (flag=(!oldalarm || time+jiffies<oldalarm))
275                 current->alarm = time+jiffies;
276             else
277                 current->alarm = oldalarm;
// 另外，如果设置了规范模式标志，那么若已读到起码一个字符则中断循环。否则若已读取数
// 大于或等于最少要求读取的字符数，则也中断循环。
278         if (L_CANON(tty)) {
279             if (b-buf)
280                 break;

```

```

281         } else if (b->buf >= minimum)
282             break;
283     }
    // 此时读取 tty 字符循环操作结束，因此让进程的定时值恢复原值。最后，如果进程接收到信
    // 号并且没有读取到任何字符，则返回出错号（被中断）。否则返回已读字符数。
284     current->alarm = oldalarm;
285     if (current->signal && !(b->buf))
286         return -EINTR;
287     return (b->buf); // 返回已读取的字符数。
288 }
289
    /// tty 写函数。
    // 把用户缓冲区中的字符写入 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。
290 int tty_write(unsigned channel, char * buf, int nr)
291 {
292     static cr_flag=0;
293     struct tty_struct * tty;
294     char c, *b=buf;
295
    // 首先判断函数参数有效性，并让 tty 指针指向参数子设备号对应 ttb_table 表中的 tty 结构。
    // 本版本 Linux 内核终端只有 3 个子设备，分别是控制台终端（0）、串口终端 1（1）和串口终
    // 端 2（2）。所以任何大于 2 的子设备号都是非法的。需要读取的字节数当然也不能小于 0。
296     if (channel>2 || nr<0) return -1;
297     tty = channel + tty_table;
    // 字符设备是一个一个字符进行处理的，所以这里对于 nr 大于 0 时对每个字符进行循环处理。
    // 在循环体中，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。
298     while (nr>0) {
299         sleep_if_full(&tty->write_q);
300         if (current->signal)
301             break;
    // 当要写的字节数 nr 还大于 0 并且 tty 写队列不满，则循环执行以下操作。首先从用户数据缓
    // 冲区中取 1 字节 c。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。
302         while (nr>0 && !FULL(tty->write_q)) {
303             c=get_fs_byte(b);
304             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）。
305                 if (c=='\r' && O_CRNL(tty))
306                     c='\n';
307                 else if (c=='\n' && O_NLRET(tty))
308                     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。
309                 if (c=='\n' && !cr_flag && O_NLCR(tty)) {
310                     cr_flag = 1;
311                     PUTCH(13, tty->write_q);
312                     continue;

```

```

313     }
314     if (0 < LCUC(tty))           // 小写转成大写字符。
315         c=toupper(c);
316 }
// 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。
317     b++; nr--;
318     cr_flag = 0;
319     PUTCH(c, tty->write_q);
320 }
// 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。
321     tty->write(tty);
322     if (nr>0)
323         schedule();
324 }
325     return (b-buf);           // 最后返回写入的字节数。
326 }
327
328 /*
329  * Jeh, sometimes I really like the 386.
330  * This routine is called from an interrupt,
331  * and there should be absolutely no problem
332  * with sleeping even in an interrupt (I hope).
333  * Of course, if somebody proves me wrong, I'll
334  * hate intel for all time :-). We'll have to
335  * be careful and see to reinstating the interrupt
336  * chips before calling this, though.
337  *
338  * I don't think we sleep here under normal circumstances
339  * anyway, which is good, as the task sleeping might be
340  * totally innocent.
341  */
/*
* 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（0，1 或 2）。
// 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（keyboard.S, 69）中被调用。
342 void do tty_interrupt(int tty)
343 {
344     copy_to_cooked(tty_table+tty);
345 }

```

```

346      // 字符设备初始化函数。空，为以后扩展做准备。
347 void chr_dev_init(void)
348 {
349 }
350

```

10.7.3 其他信息

10.7.3.1 控制字符 VTIME、VMIN

在非规范模式下，这两个值是超时定时值和最小读取字符个数。MIN 表示为了满足读操作，需要读取的最少字符数。TIME 是一个十分之一秒计数的超时计时值。当这两个都设置的话，读操作将等待，直到至少读到一个字符，如果在超时之前收到了 MIN 个字符，则读操作即被满足。如果在 MIN 个字符被收到之前就已超时，就将到此时已收到的字符返回给用户。如果仅设置了 MIN，那么在读取 MIN 个字符之前读操作将不返回。如果仅设置了 TIME，那么在读到至少一个字符或者定时超时后读操作将立刻返回。如果两个都没有设置，则读操作将立刻返回，仅给出目前已读的字节数。详细说明参见 `termios.h` 文件。

10.8 tty_ioctl.c 程序

10.8.1 功能描述

本文件用于字符设备的控制操作，实现了函数 `tty_ioctl()`。程序通过使用该函数可以修改指定终端 `termios` 结构中的设置标志等信息。`tty_ioctl()` 函数将由 `fs/ioctl.c` 中的输入输出控制系统调用 `sys_ioctl()` 来调用。

10.8.2 代码注释

程序 10-7 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
18 // 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。
19 // 例如波特率是 2400bps 时，对应的因子是 48 (0x30)；9600bps 的因子是 12 (0x1c)。

```

```

18 static unsigned short quotient[] = {
19     0, 2304, 1536, 1047, 857,
20     768, 576, 384, 192, 96,
21     64, 48, 24, 12, 6, 3
22 };
23
24 // 修改传输波特率。
25 // 参数: tty - 终端对应的 tty 数据结构。
26 // 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
27 // 字节和高字节。写完后复位 DLAB 位。对于串口 2, 这两个端口分别是 0x2f8 和 0x2f9。
28 static void change_speed(struct tty_struct * tty)
29 {
30     unsigned short port, quot;
31
32     // 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
33     // 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
34     // 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
35     // 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。
36     // CBAUD 是控制模式标志集中波特率位屏蔽码。
37     if (!(port = tty->read_q.data))
38         return;
39     quot = quotient[tty->termios.c_cflag & CBAUD];
40     // 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
41     // 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
42     // 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)。
43     // 最后再复位 LCR 的 DLAB 标志位。
44     cli();
45     outb_p(0x80, port+3);    /* set DLAB */ // 首先设置除数锁定标志 DLAB。
46     outb_p(quot & 0xff, port); /* LS of divisor */ // 输出因子低字节。
47     outb_p(quot >> 8, port+1); /* MS of divisor */ // 输出因子高字节。
48     outb(0x03, port+3);    /* reset DLAB */ // 复位 DLAB。
49     sti();
50 }
51
52 // 刷新 tty 缓冲队列。
53 // 参数: queue - 指定的缓冲队列指针。
54 // 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。
55 static void flush(struct tty_queue * queue)
56 {
57     cli();
58     queue->head = queue->tail;
59     sti();
60 }
61
62 // 等待字符发送出去。
63 static void wait_until_sent(struct tty_struct * tty)
64 {
65     /* do nothing - not implemented */ /* 什么都没做 - 还未实现 */
66 }
67
68 // 发送 BREAK 控制符。
69 static void send_break(struct tty_struct * tty)
70 {

```



```

53      /* do nothing - not implemented */ /* 什么都没做 - 还未实现 */
54  }
55
56      //// 取终端 termios 结构信息。
57      // 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。
58      static int get_termios(struct tty_struct * tty, struct termios * termios)
59  {
60      int i;
61
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
63      // 的 termios 结构信息到用户缓冲区中。最后返回 0。
64      verify_area(termios, sizeof (*termios));          // kernel/fork.c, 24 行。
65      for (i=0 ; i< (sizeof (*termios)) ; i++)
66          put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
67      return 0;
68  }
69
70      //// 设置终端 termios 结构信息。
71      // 参数: tty - 指定终端的 tty 结构指针; termios - 用户数据区 termios 结构指针。
72      static int set_termios(struct tty_struct * tty, struct termios * termios)
73  {
74      int i;
75
76      // 首先把用户数据区中 termios 结构信息复制到指定终端 tty 结构的 termios 结构中。因为用
77      // 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
78      // 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。
79      for (i=0 ; i< (sizeof (*termios)) ; i++)
80          ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
81      change_speed(tty);
82      return 0;
83  }
84
85      //// 读取 termio 结构中的信息。
86      // 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。
87      static int get_termio(struct tty_struct * tty, struct termio * termio)
88  {
89      int i;
90      struct termio tmp_termio;
91
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
93      // 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
94      // 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
95      // 中目的是为了进行数据类型转换。
96      verify_area(termio, sizeof (*termio));
97      tmp_termio.c_iflag = tty->termios.c_iflag;
98      tmp_termio.c_oflag = tty->termios.c_oflag;
99      tmp_termio.c_cflag = tty->termios.c_cflag;
100     tmp_termio.c_lflag = tty->termios.c_lflag;
101     tmp_termio.c_line = tty->termios.c_line;
102     for(i=0 ; i < NCC ; i++)
103         tmp_termio.c_cc[i] = tty->termios.c_cc[i];
104     // 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。
105     for (i=0 ; i< (sizeof (*termio)) ; i++)

```



```

90         put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
91     return 0;
92 }
93
94 /*
95  * This only works as the 386 is low-byt-first
96  */
97 /*
98  * 下面 termio 设置函数仅适用于低字节在前的 386 CPU。
99  */
100 // 设置终端 termio 结构信息。
101 // 参数: tty - 指定终端的 tty 结构指针; termio - 用户数据区中 termio 结构。
102 // 将用户缓冲区 termio 的信息复制到终端的 termios 结构中。返回 0 。
103 static int set_termio(struct tty_struct * tty, struct termio * termio)
104 {
105     int i;
106     struct termio tmp_termio;
107
108     // 首先复制用户数据区中 termio 结构信息到临时 termio 结构中。然后再将 termio 结构的信息
109     // 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
110     // 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
111     // 是完全相同的。
112     for (i=0 ; i< (sizeof (*termio)) ; i++)
113         ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
114     *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
115     *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
116     *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
117     *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
118     tty->termios.c_line = tmp_termio.c_line;
119     for(i=0 ; i < NCC ; i++)
120         tty->termios.c_cc[i] = tmp_termio.c_cc[i];
121     // 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
122     // 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。
123     change_speed(tty);
124     return 0;
125 }
126
127 // 终端设备输入输出控制函数。
128 // 参数: dev - 设备号; cmd - ioctl 命令; arg - 操作参数指针。
129 // 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
130 // 处理。
131 int tty_ioctl(int dev, int cmd, int arg)
132 {
133     struct tty_struct * tty;
134     // 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),
135     // 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
136     // 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,
137     // 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)、
138     // 2 (串口 2 终端)。
139     if (MAJOR(dev) == 5) {
140         dev=current->tty;
141         if (dev<0)
142             panic("tty_ioctl: dev<0");

```

```

122     } else
123         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。
124     tty = dev + tty\_table;
125     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。
126         case TCGETS:
127             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。紧接着执行下面设置终端 termios 结构的操作。
128         case TCSETSF:
129             flush(&tty->read_q); /* fallback */ /* 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。
130         case TCSETSW:
131             wait\_until\_sent(tty); /* fallback */ /* 接着继续执行 */
    // 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。
132         case TCSETS:
133             return set\_termios(tty, (struct termios *) arg);
    // 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。
134         case TCGETA:
135             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。紧接着执行下面设置终端 termio 结构的操作。
136         case TCSETAF:
137             flush(&tty->read_q); /* fallback */
    // 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。
138         case TCSETAW:
139             wait\_until\_sent(tty); /* fallback */
    // 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。
140         case TCSETA:
141             return set\_termio(tty, (struct termio *) arg);
    // 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。
142         case TCSBRK:
143             if (!arg) {
144                 wait\_until\_sent(tty);
145                 send\_break(tty);
146             }
147             return 0;
    // 开始/停止流控制。如果参数值是 0，则挂起输出；如果是 1，则恢复挂起的输出；如果是 2，
    // 则挂起输入；如果是 3，则重新开启挂起的输入。
148         case TCXONC:
149             return -EINVAL; /* not implemented */ /* 未实现 */
    // 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。
150         case TCFLSH:
151             if (arg==0)
152                 flush(&tty->read_q);
153             else if (arg==1)
154                 flush(&tty->write_q);
155             else if (arg==2) {

```

```

156             flush(&tty->read_q);
157             flush(&tty->write_q);
158         } else
159             return -EINVAL;
160         return 0;
// 设置终端串行线路专用模式。
161         case TIOCEXCL:
162             return -EINVAL; /* not implemented */ /* 未实现 */
// 复位终端串行线路专用模式。
163         case TIOCNXCL:
164             return -EINVAL; /* not implemented */ /* 未实现 */
// 设置 tty 为控制终端。(TIOCNOTTY - 禁止 tty 为控制终端)。
165         case TIOCSCTTY:
166             return -EINVAL; /* set controlling term NI */ /* 未实现 */
// 读取终端设备进程组号。首先验证用户缓冲区长度，然后复制 tty 的 pgrp 字段到用户缓冲区。
// 此时参数 arg 是用户缓冲区指针。
167         case TIOCGPGRP:
168             verify_area((void *) arg, 4);
169             put_fs_long(tty->pgrp, (unsigned long *) arg);
170             return 0;
// 设置终端设备的进程组号 pgrp。此时参数 arg 是用户缓冲区中 pgrp 的指针。
171         case TIOCSPGRP:
172             tty->pgrp=get_fs_long((unsigned long *) arg);
173             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。
174         case TIOCOUTQ:
175             verify_area((void *) arg, 4);
176             put_fs_long(CHARS(tty->write_q), (unsigned long *) arg);
177             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。
178         case TIOCINQ:
179             verify_area((void *) arg, 4);
180             put_fs_long(CHARS(tty->secondary),
181                         (unsigned long *) arg);
182             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。
183         case TIOCSSTI:
184             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）。
185         case TIOCGWINSZ:
186             return -EINVAL; /* not implemented */ /* 未实现 */
// 设置终端设备窗口大小信息（参见 winsize 结构）。
187         case TIOCSWINSZ:
188             return -EINVAL; /* not implemented */ /* 未实现 */
// 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）。
189         case TIOCMGET:
190             return -EINVAL; /* not implemented */ /* 未实现 */
// 设置单个 modem 状态控制引线的状态（true 或 false）。
191         case TIOCMBIS:
192             return -EINVAL; /* not implemented */ /* 未实现 */

```

```

193 // 复位单个 MODEM 状态控制引线的状态。
194 case TIOCMBIC:
195     return -EINVAL; /* not implemented */ /* 未实现 */
196 // 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。
197 case TIOCMSET:
198     return -EINVAL; /* not implemented */ /* 未实现 */
199 // 读取软件载波检测标志（1 - 开启；0 - 关闭）。
200 case TIOCGSOFTCAR:
201     return -EINVAL; /* not implemented */ /* 未实现 */
202 // 设置软件载波检测标志（1 - 开启；0 - 关闭）。
203 case TIOCSSOFTCAR:
204     return -EINVAL; /* not implemented */ /* 未实现 */
205 default:
206     return -EINVAL;
207 }
208 }
```

10.8.3 其他信息

10.8.3.1 波特率与波特率因子

波特率 = 1.8432MHz / (16 X 波特率因子)。常用波特率与波特率因子的对应关系见表 10-9 所示。



表 10-9 波特率与波特率因子对应表

波特率	波特率因子		波特率	波特率因子	
	MSB,LSB	合并值		MSB,LSB	合并值
50	0x09,0x00	2304	1200	0x00,0x60	96
75	0x06,0x00	1536	1800	0x00,0x40	64
110	0x04,0x17	1047	2400	0x00,0x30	48
134.5	0x03,0x59	857	4800	0x00,0x18	24
150	0x03,0x00	768	9600	0x00,0x1c	12
200	0x02,0x40	576	19200	0x00,0x06	6
300	0x01,0x80	384	38400	0x00,0x03	3
600	0x00,0xc0	192			

第11章 数学协处理器(math)

内核目录 kernel/math 目录下包含数学协处理器仿真处理代码文件，但该程序目前还没有真正实现数学协处理器的仿真代码，仅含有一个程序外壳，见列表 11-1 所示。

列表 11-1 linux/kernel/math 目录

	名称	大小	最后修改时间 (GMT)	说明
	Makefile	936 bytes	1991-11-18 00:21:45	
	math_emulate.c	1023 bytes	1991-11-23 15:36:34	

11.1 Makefile 文件

11.1.1 功能描述

math 目录下程序的编译管理文件。

11.1.2 代码注释

程序 11-1 linux/kernel/math/Makefile

```

1 #
2 # Makefile for the FREAX-kernel character device drivers.
3 #
4 # Note! Dependencies are done automagically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
8 # FREAX(Linux)内核字符设备驱动程序的 Makefile 文件。
9 # 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
10 # 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c文件的信息）。
11
12 AR      =gar      # 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。
13 AS      =gas      # GNU 的汇编程序。
14 LD      =gld      # GNU 的连接程序。
15 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
16 CC      =gcc      # GNU C 语言编译器。
17 # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
18 # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
19 # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
20 # 单短小的函数代码嵌入调用程序中；-mstring-insns Linux 自己添加的优化选项，以后不再使用；
21 # -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(.././include)。
22 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
23         -finline-functions -mstring-insns -nostdinc -I.././include
24 # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输

```

```

# 出设备或指定的输出文件中; -nostdinc -I.././include 同前。
16 CPP      =gcc -E -nostdinc -I.././include
17
# 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
# 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止 (-S), 从而产生与
# 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
# 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。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
# 用第一个规则; 若目标是.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。
18 .c.s:
19     $(CC) $(CFLAGS) \
20     -S -o $.s $<
# 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22     $(AS) -c -o $.o $<
23 .c.o:                                # 类似上面, *.c 文件->*.o 目标文件。不进行连接。
24     $(CC) $(CFLAGS) \
25     -c -o $.o $<
26
27 OBJS = math_emulate.o    # 定义目标文件变量 OBJS。
28
# 在有了先决条件 OBJS 后使用下面的命令连接成目标 math.a 库文件。
# 命令行中的 'rcs' 是操作码和修饰标志 (前面可加上'-'), 放置次序任意。其中 'r' 是操作码,
# 说明需要执行的操作。'r' 表示要把命令行末列出的目标文件插入 (替换 replacement) 归档文件
# blk_drv.a 中。'cs' 是两个修饰标志, 用于修饰具体操作行为。'c' 表示当归档文件 blk_drv.a 不
# 存在时就创建这个文件。's' 表示写进或更新归档文件中的目标文件索引。 对一个归档文件单独
# 使用命令 "ar s" 等同于对一个归档文件执行命令 ranlib。
29 math.a: $(OBJS)
30     $(AR) rcs math.a $(OBJS)
31     sync
32
# 下面的规则用于清理工作。当执行'make clean'时, 就会执行下面的命令, 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。
33 clean:
34     rm -f core *.o *.a tmp_make
35     for i in *.c;do rm -f `basename $$i .c`.s;done
36
# 下面得目标或规则用于检查各文件之间的依赖关系。方法如下:
# 使用字符串编辑程序 sed 对 Makefile 文件 (即是本文件) 进行处理, 输出为删除 Makefile
# 文件中'### Dependencies' 行后面的所有行, 并生成 tmp_make 临时文件。然后对 kernel/math/
# 目录下的每个 C 文件执行 gcc 预处理操作。
# -M 标志告诉预处理程序输出描述每个目标文件相关性的规则, 并且这些规则符合 make 语法。
# 对于每一个源文件, 预处理程序输出一个 make 规则, 其结果形式是相应源程序文件的目标
# 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
# 文件 tmp_make 中, 然后将该临时文件复制成新的 Makefile 文件。
37 dep:
38     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
39     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,\'` " "; \
40         $(CPP) -M $$i;done) >> tmp_make
41     cp tmp_make Makefile
42

```

[43](#) ### Dependencies:

11.2 math-emulation.c 程序

11.2.1 功能描述

数学协处理器仿真处理代码文件。该程序目前还没有实现对数学协处理器的仿真代码。仅实现了协处理器发生异常中断时调用的两个 C 函数。math_emulate() 仅在用户程序中包含协处理器指令时，对进程设置协处理器异常信号。

11.2.2 代码注释

程序 11-2 linux/kernel/math/math_emulate.c

```

1  /*
2   * linux/kernel/math/math_emulate.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * This directory should contain the math-emulation code.
9   * Currently only results in a signal.
10  */
11  /*
12   * 该目录里应该包含数学仿真代码。目前仅产生一个信号。
13   */
14
15  #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。
16
17  #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，
18                               // 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。
19
20  #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
21
22  #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。
23
24  // 协处理器仿真函数。
25  // 中断处理程序调用的 C 函数，参见 (kernel/math/system_call.s, 169 行)。
26  // 第 22 行上两个参数的位置应该对换。
27
28  void math_emulate(long edi, long esi, long ebp, long sys_call_ret,
29                    long eax, long ebx, long ecx, long edx,
30                    unsigned short fs, unsigned short es, unsigned short ds,
31                    unsigned long eip, unsigned short cs, unsigned long eflags,
32                    unsigned short ss, unsigned long esp)
33  {
34      unsigned char first, second;
35
36      /* 0x0007 means user code space */
37      /* 0x0007 表示用户代码空间 */
38      // 选择符 0x000F 表示在局部描述符表中描述符索引值=1，即代码空间。如果段寄存器 cs 不等于 0x000F
39      // 则表示 cs 一定是内核代码选择符，是在内核代码空间，则出错，显示此时的 cs:eip 值，并显示信息

```

```
// “内核中需要数学仿真”，然后进入死机状态。
27     if (cs != 0x000F) {
28         printk("math_emulate: %04x:%08x\n|r", cs, eip);
29         panic("Math emulation needed in kernel");
30     }
// 取用户进程进入中断时当前代码指针 cs:eip 处的两个字节机器码 first 和 second。是这个指令引发
// 了本中断。然后显示进程的 cs、eip 值和这两个字节，并给进程设置浮点异常信号 SIGFPE。
31     first = get_fs_byte((char *)(&eip));
32     second = get_fs_byte((char *)(&eip));
33     printk("%04x:%08x %02x %02x\n|r", cs, eip-2, first, second);
34     current->signal |= 1<<(SIGFPE-1);
35 }
36
//// 协处理器出错处理函数。
// 中断处理程序调用的 C 函数，参见(kernel/math/system_call.s, 145 行)。
37 void math_error(void)
38 {
// 协处理器指令。(以非等待形式)清除所有异常标志、忙标志和状态字位 7。
39     __asm__("fnclex");
// 如果上个任务使用过协处理器，则向上个任务发送协处理器异常信号。
40     if (last_task_used_math)
41         last_task_used_math->signal |= 1<<(SIGFPE-1);
42 }
43
```

第12章 文件系统(fs)

本章涉及 linux 内核中文件系统的实现代码和用于块设备的高速缓冲区管理程序。在开发 Linux 0.11 内核文件系统时, Linus 主要参照了 Andrew S.Tanenbaum 著的《MINIX 操作系统设计与实现》一书, 使用了其中 1.0 版的 MINIX 文件系统。因此在阅读本章内容时, 可以参考该书有关 MINIX 文件系统的相关章节。而高速缓冲区的工作原理可参见 M.J.Bach 的《UNIX 操作系统设计》第三章内容。

列表 12-1 linux/fs 目录

名称	大小	最后修改时间 (GMT)	说明
 Makefile	5053 bytes	1991-12-02 03:21:31	m
 bitmap.c	4042 bytes	1991-11-26 21:31:53	m
 block_dev.c	1422 bytes	1991-10-31 17:19:55	m
 buffer.c	9072 bytes	1991-12-06 20:21:00	m
 char_dev.c	2103 bytes	1991-11-19 09:10:22	m
 exec.c	9134 bytes	1991-12-01 20:01:01	m
 fcntl.c	1455 bytes	1991-10-02 14:16:29	m
 file_dev.c	1852 bytes	1991-12-01 19:02:43	m
 file_table.c	122 bytes	1991-10-02 14:16:29	m
 inode.c	6933 bytes	1991-12-06 20:16:35	m
 ioctl.c	977 bytes	1991-11-19 09:13:05	
 namei.c	16562 bytes	1991-11-25 19:19:59	m
 open.c	4340 bytes	1991-11-25 19:21:01	m
 pipe.c	2385 bytes	1991-10-18 19:02:33	m
 read_write.c	2802 bytes	1991-11-25 15:47:20	m
 stat.c	1175 bytes	1991-10-02 14:16:29	m
 super.c	5628 bytes	1991-12-06 20:10:12	m
 truncate.c	1148 bytes	1991-10-02 14:16:29	m

12.1 总体功能

本章所注释说明的程序量较大, 但是通过第 2 章中对 Linux 源代码目录结构的分析 (参见图 2-28 fs 目录中各程序中函数之间的引用关系), 我们可以把它们从功能上分为四个部分来加以讨论。第一部分是有关高速缓冲区的管理程序, 主要实现了对硬盘等块设备进行数据高速存取的函数。该部分内容集中在 `buffer.c` 程序中实现; 第二部分代码描述了文件系统的低层通用函数。说明了文件索引节点的管理、磁盘数据块的分配和释放以及文件名与 i 节点的转换算法; 第三部分程序是有关对文件中数据进行读写操作, 包括对字符设备、管道、块读写文件中数据的访问; 第四部分的程序主要涉及文件的系统调用接口的实现, 主要涉及文件打开、关闭、创建以及有关文件目录操作等的系统调用。

下面首先介绍一下 MINIX 文件系统的基本结构，然后分别对这四部分加以说明。

12.1.1 MINIX 文件系统

目前 MINIX 的版本是 2.0，所使用的文件系统是 2.0 版，它与其 1.5 版系统之前的版本不同，对其容量已经作了扩展。但由于本书注释的 linux 内核使用的是 MINIX 文件系统 1.0 版本，所以这里仅对其 1.0 版文件系统作简单介绍。

MINIX 文件系统与标准 UNIX 的文件系统基本相同。它由 6 个部分组成。对于一个 360K 的软盘，其各部分的分布见图 12-1 所示。

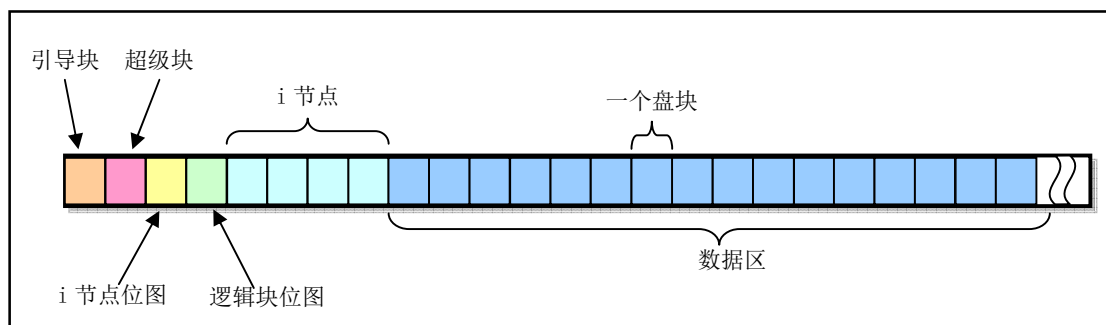


图 12-1 建有 MINIX 文件系统的一个 360K 软盘中文件系统各部分的布局示意图

图中，整个磁盘被划分成以 1KB 为单位的磁盘块，因此上图中共有 360 个磁盘块，每个方格表示一个磁盘块。在后面的说明我们会知道，在 MINIX 1.0 文件系统中，其磁盘块大小与逻辑块大小正好是一样的，也是 1KB 字节。因此 360KB 盘片也含有 360 个逻辑块。在后面的讨论中我们有时会混合使用这两个名称。

引导块是计算机加电启动时可由 ROM BIOS 自动读入的执行代码和数据。但并非所有盘都用于作为引导设备，所以对于不用于引导的盘片，这一盘块中可以不含代码。但任何盘片必须含有引导块空间，以保持 MINIX 文件系统格式的统一。即文件系统只是在块设备上空出一个存放引导块的空间。如果你把内核映像文件放在文件系统中，那么你就可以在文件系统所在设备的第 1 个块（即引导块空间）存放实际的引导程序，并由它来取得和加载文件系统上的内核映像文件。

对于硬盘块设备，通常在其上会划分出几个分区，并且在每个分区中都可存放一个不同的完整文件系统，见图 12-2 所示。图中表示有 4 个分区，分别存放着 FAT32 文件系统、NTFS 文件系统、MINIX 文件系统和 EXT2 文件系统。硬盘的第一个扇区是主引导扇区，其中存放着硬盘引导程序和分区表信息。分区表中的信息指明了硬盘上每个分区的类型、在硬盘中起始位置参数和结束位置参数以及占用的扇区总数，参见 kernel/blk_drv/hd.c 文件后的硬盘分区表结构。

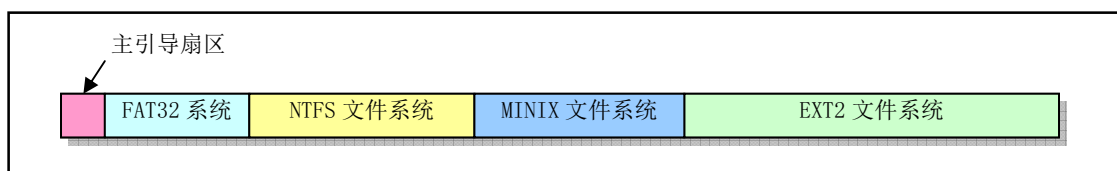


图 12-2 硬盘设备上的分区和文件系统

超级块用于存放盘设备上文件系统结构的信息，并说明各部分的大小。其结构见图 12-3 所示。其中，s_ninodes 表示设备上的 i 节点总数。s_nzones 表示设备上以逻辑块为单位的总逻辑块数。s_imap_blocks 和 s_zmap_blocks 分别表示 i 节点位图和逻辑块位图所占用的磁盘块数。s_firstdatazone 表示设备上数据

区开始处占用的第一个逻辑块块号。`s_log_zone_size` 是使用 2 位底的对数表示的每个逻辑块包含的磁盘块数。对于 MINIX 1.0 文件系统该值为 0，因此其逻辑块的大小就等于磁盘块大小，都是 1KB。`s_max_size` 是以字节表示的最大文件长度。当然这个长度值将受到磁盘容量的限制。`s_magic` 是文件系统魔幻数，用以指明文件系统的类型。对于 MINIX 1.0 文件系统，它的魔幻数是 0x137f。

在 Linux 0.11 系统中，被加载的文件系统超级块保存在超级块表（数组）`super_block[]` 中。该表共有 8 项，因此 Linux 0.11 系统中同时最多加载 8 个文件系统。超级块表将在 `super.c` 程序的 `mount_root()` 函数中被初始化，在 `read_super()` 函数中会为新加载的文件系统在表中设置一个超级块项，并在 `put_super()` 函数中释放超级块表中指定的超级块项。

		字段名称	数据类型	说明
出现在盘上和内存中的字段	{	<code>s_ninodes</code>	short	i 节点数
		<code>s_nzones</code>	short	逻辑块数(或称为区块数)
		<code>s_imap_blocks</code>	short	i 节点位图所占块数
		<code>s_zmap_blocks</code>	short	逻辑块位图所占块数
		<code>s_firstdatazone</code>	short	数据区中第一个逻辑块块号
		<code>s_log_zone_size</code>	short	\log_2 (磁盘块数/逻辑块)
		<code>s_max_size</code>	long	最大文件长度
		<code>s_magic</code>	short	文件系统幻数 (0x137f)
仅在内存中使用的字段	{	<code>s_imap[8]</code>	buffer_head *	i 节点位图在高速缓冲块指针数组
		<code>s_zmap[8]</code>	buffer_head *	逻辑块位图在高速缓冲块指针数组
		<code>s_dev</code>	short	超级块所在设备号
		<code>s_isup</code>	m_inode *	被安装文件系统根目录 i 节点
		<code>s_imount</code>	m_inode *	该文件系统被安装到的 i 节点
		<code>s_time</code>	long	修改时间
		<code>s_wait</code>	task_struct *	等待本超级块的进程指针
		<code>s_lock</code>	char	锁定标志
		<code>s_rd_only</code>	char	只读标志
		<code>s_dirt</code>	char	已被修改(脏)标志

图 12-3 MINIX 的超级块结构

逻辑块位图用于描述盘上每个数据盘块的使用情况。除第 1 个比特位（位 0）以外，逻辑块位图中每个比特位依次代表盘上数据区中的一个逻辑块。因此逻辑块位图的比特位 1 代表盘上数据区中第一个数据盘块，而非盘上的第一个磁盘块（引导块）。当一个数据盘块被占用时，则逻辑块位图中相应比特位被置位。由于当所有磁盘数据盘块都被占用时查找空闲盘块的函数会返回 0 值，因此逻辑块位图最低比特位（位 0）闲置不用，并且在创建文件系统时会预先将其设置为 1。

从超级块的结构中我们还可以看出，逻辑块位图最多使用 8 块缓冲块（`s_zmap[8]`），而每块缓冲块大小是 1024 字节，每比特表示一个盘块的占用状态，因此一个缓冲块可代表 8192 个盘块。8 个缓冲块总共可表示 65536 个盘块，因此 MINIX 文件系统 1.0 所能支持的最大块设备容量（长度）是 64MB。

i 节点位图用于说明 i 节点是否被使用，同样是每个比特位代表一个 i 节点。对于 1K 大小的盘块来讲，一个盘块就可表示 8192 个 i 节点的使用状况。与逻辑块位图的情况类似，由于当所有 i 节点都被使用时查找空闲 i 节点的函数会返回 0 值，因此 i 节点位图第 1 个字节的最低比特位（位 0）和对应的 i 节点 0 都闲置不用，并且在创建文件系统时会预先将 i 节点 0 对应比特位图中的比特位置为 1。因此第一个 i 节点位图块中只能表示 8191 个 i 节点的状况。

盘上的 i 节点部分存放着文件系统中文件或目录名的索引节点，每个文件或目录名都有一个 i 节点。每个 i 节点结构中存放着对应文件的相关信息，如文件宿主的 id(uid)、文件所属组 id(gid)、文件长度、访问修改时间以及文件数据块在盘上的位置等。整个结构共使用 32 个字节，见图 12-4 所示。

		字段名称	数据类型	说明
在盘上和内存中的 字段，共 32 字节	{	i_mode	short	文件的类型和属性（ <code>rwX</code> 位）
		i_uid	short	文件宿主的用户 id
		i_size	long	文件长度（字节）
		i_mtime	long	修改时间（从 1970.1.1:0 时算起，秒）
		i_gid	char	文件宿主的组 id
		i_nlinks	char	链接数（有多少个文件目录项指向该 i 节点）
		i_zone[9]	short	文件所占用的盘上逻辑块号数组。其中： zone[0]-zone[6]是直接块号； zone[7]是一次间接块号； zone[8]是二次（双重）间接块号。 注：zone 是区的意思，可译成区块或逻辑块。 对于设备特殊文件名的 i 节点，其 zone[0]中存放的是该文件名所指设备的设备号。
仅在内存中使用的 字段	{	i_wait	task_struct *	等待该 i 节点的进程。
		i_atime	long	最后访问时间。
		i_ctime	long	i 节点自身被修改时间。
		i_dev	short	i 节点所在的设备号。
		i_num	short	i 节点号。
		i_count	short	i 节点被引用的次数，0 表示空闲。
		i_lock	char	i 节点被锁定标志。
		i_dirt	char	i 节点已被修改（脏）标志。
		i_pipe	char	i 节点用作管道标志。
		i_mount	char	i 节点安装了其他文件系统标志。
		i_seek	char	搜索标志（ <code>lseek</code> 操作时）。
		i_update	char	i 节点已更新标志。

图 12-4 MINIX 文件系统 1.0 版的 i 节点结构

`i_mode` 字段用来保存文件的类型和访问权限属性。其比特位 15-12 用于保存文件类型，位 11-9 保存执行文件时设置的信息，位 8-0 表示文件的访问权限，见图 12-5 所示。具体信息参见文件 `include/sys/stat.h` 第 20—50 行和 `include/fcntl.h`。

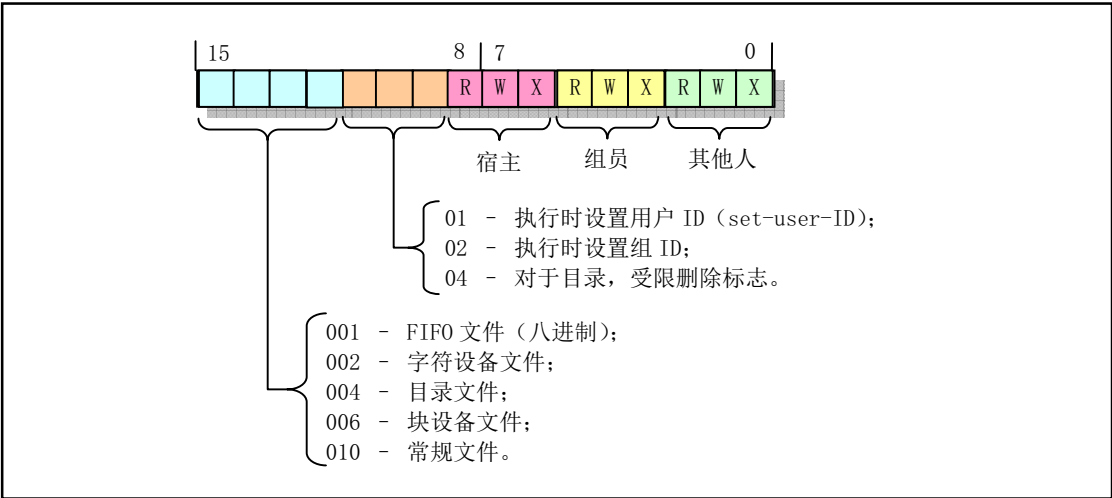


图 12-5 i 节点属性字段内容

文件中的数据是放在磁盘块的数据区中的，而一个文件名则通过对应的 *i* 节点与这些数据磁盘块相联系，这些盘块的号码就存放在 *i* 节点的逻辑块数组 *i_zone* 中。其中，*i_zone* 数组用于存放 *i* 节点对应文件的盘块号。*i_zone*[0] 到 *i_zone*[6] 用于存放文件开始的 7 个磁盘块号，称为直接块。若文件长度小于等于 7K 字节，则根据其 *i* 节点可以很快就找到它所使用的盘块。若文件大一些时，就需要用到一次间接块了 (*i_zone*[7])，这个盘块中存放着附加的盘块号。对于 MINIX 文件系统它可以存放 512 个盘块号，因此可以寻址 512 个盘块。若文件还要大，则需要使用二次间接盘块 (*i_zone*[8])。二次间接块的一级盘块的作用类似与一次间接盘块，因此使用二次间接盘块可以寻址 512*512 个盘块。参见图 12-6 所示。

另外，对于 /dev/ 目录下的设备文件来说，它们并不占用磁盘数据区中的数据盘块，即它们文件的长度是 0。设备文件名的 *i* 节点仅用于保存其所定义设备的属性和设备号。设备号被存放在设备文件 *i* 节点的 *zone*[0] 中。

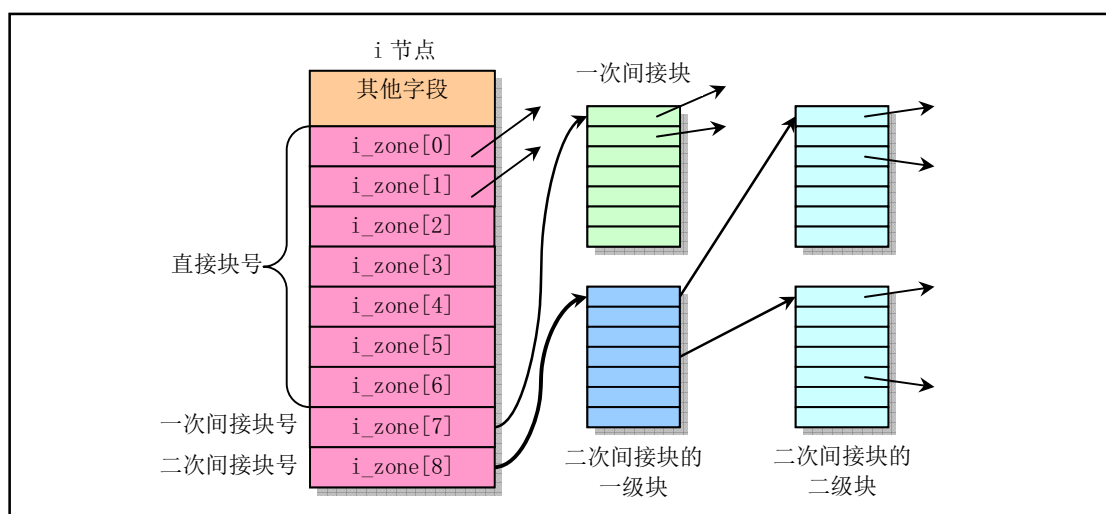


图 12-6 *i* 节点的逻辑块（区块）数组的功能

当所有 *i* 节点都被使用时，查找空闲 *i* 节点的函数会返回值 0，因此 *i* 节点位图最低比特位和 *i* 节点 0 都不使用。*i* 节点 0 的结构被初始化成全零，并在创建文件系统时将 *i* 节点 0 的比特位置位。

对于 PC 机来讲，一般以一个扇区的长度（512 字节）作为块设备的数据块长度。而 MINIX 文件系统则将连续的 2 个扇区数据（1024 字节）作为一个数据块来处理，称之为一个磁盘块或盘块。其长度与高速缓冲区中的缓冲块长度相同。编号是从盘上第一个盘块开始算起，也即引导块是 0 号盘块。而上述的逻辑块或区块，则是盘块的 2 的幂次倍数。一个逻辑块长度可以等于 1、2、4 或 8 个盘块长度。对于本书所讨论的 linux 内核，逻辑块的长度等于盘块长度。因此在代码注释中这两个术语含义相同。但是术语数据逻辑块（或数据盘块）则是指盘设备上数据部分中，从第一个数据盘块开始编号的盘块。

12.1.2 文件类型、属性和目录项

12.1.2.1 文件的类型和属性

UNIX 类操作系统中的文件通常可分为 6 类。如果在 shell 下执行 "ls -l" 命令，我们就可以从所列出的文件状态信息中知道文件的类型。见图 12-7 所示。

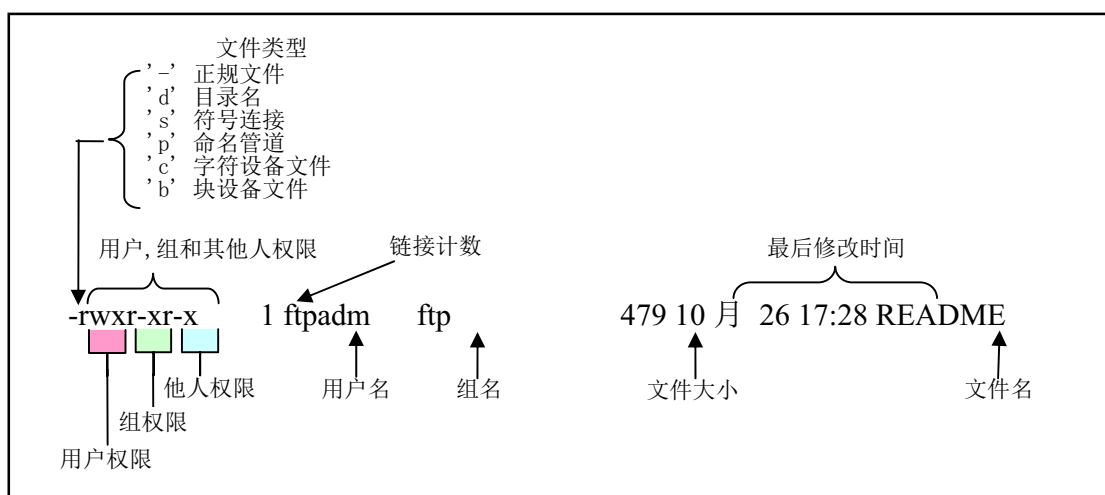


图 12-7 命令 'ls -l' 显示的文件信息

图中, 命令显示的第一个字节表示所列文件的类型。'-'表示该文件是一个正规(一般)文件。

正规文件('-')是一类文件系统对其不作解释的文件, 包含有任何长度的字节流。例如源程序文件、二进制执行文件、文档以及脚本文件。

目录('d')在 UNIX 文件系统中也是一种文件, 但文件系统管理会对其内容进行解释, 以使人们可以看到有那些文件包含在一个目录中, 以及它们是如何组织在一起构成一个分层次的文件系统的。

符号连接('s')用于使用一个不同文件名来引用另一个文件。符号连接可以跨越一个文件系统而连接到另一个文件系统中的文件上。删除一个符号连接并不影响被连接的文件。另外还有一种连接方式称为“硬连接”。它与这里所说符号连接中被连接文件的地位相同, 被作为一般文件对待, 但不能跨越文件系统(或设备)进行连接, 并且会递增文件的连接计数值。见下面对链接计数的说明。

命名管道('p')文件是系统创建有名管道时建立的文件。可用于无关进程之间的通信。

字符设备('c')文件用于以操作文件的方式访问字符设备, 例如 tty 终端、内存设备以及网络设备。

块设备('b')文件用于访问象硬盘、软盘等的设备。在 UNIX 类操作系统中, 块设备文件和字符设备文件一般均存放在系统的/dev 目录中。

在 linux 内核中, 文件的类型信息保存在对应 i 节点的 i_mode 字段中, 使用高 4 比特位来表示, 并使用了一些判断文件类型宏, 例如 S_ISBLK、S_ISDIR 等, 这些宏在 include/sys/stat.h 中定义。

在图中文件类型字符后面是每三个字符一组构成的三组文件权限属性。用于表示文件宿主、同组用户和其他用户对文件的访问权限。'rwx'分别表示对文件可读、可写和可执行的许可权。对于目录文件, 可执行表示可以进入目录。在对文件的权限进行操作时, 一般使用八进制来表示它们。例如'755'表示文件宿主对文件可以读/写/执行, 同组用户和其他人可以读和执行文件。在 linux 0.11 源代码中, 文件权限信息也保存在对应 i 节点的 i_mode 字段中, 使用该字段的低 9 比特位表示三组权限。并常使用变量 mode 来表示。有关文件权限的宏在 include/fcntl.h 中定义。

图中的'链接计数'位表示该文件被硬连接引用的次数。当计数减为零时, 该文件即被删除。'用户名'表示该文件宿主的名称, '组名'是该用户所属组的名称。

12.1.2.2 文件系统目录项结构

Linux 0.11 系统采用的是 MINIX 文件系统 1.0 版。它的目录结构和目录项结构与传统 UNIX 文件的目录项结构相同, 定义在 include/linux/fs.h 文件中。在文件系统的一个目录中, 其中所有文件名信息对应的目录项存储在该目录文件名文件的数据块中。例如, 目录名 root/下的所有文件名的目录项就保存在 root/目录名文件的数据块中。而文件系统根目录下的所有文件名信息则保存在指定 i 节点(即 1 号 i 节点)的数据块中。文件名目录项结构见如下所示:

```
// 定义在 include/linux/fs.h 文件中。
#define NAME_LEN 14 // 名字长度值。
#define ROOT_INO 1 // 根 i 节点。

// 文件目录项结构。
struct dir_entry {
    unsigned short inode; // i 节点号。
    char name[NAME_LEN]; // 文件名。
};
```

每个目录项只包括一个长度为 14 字节的文件名字符串和该文件名对应的 2 字节的 i 节点号。因此一个逻辑磁盘块可以存放 $1024/16=64$ 个目录项。有关文件的其它信息则被保存在该 i 节点号指定的 i 节点结构中，该结构中主要包括文件访问属性、宿主、长度、访问保存时间以及所在磁盘块等信息。每个 i 节点号的 i 节点都位于磁盘上的固定位置处。

在打开一个文件时，文件系统会根据给定的文件名找到其 i 节点号，从而通过其对应 i 节点信息找到文件所在的磁盘块位置，见图 12-8 所示。例如对于要查找文件名 `/usr/bin/vi` 的 i 节点号，文件系统首先会从具有固定 i 节点号（1）的根目录开始操作，即从 i 节点号 1 的数据块中查找到名称为 `usr` 的目录项，从而得到文件 `/usr` 的 i 节点号。根据该 i 节点号文件系统可以顺利地取得目录 `/usr`，并在其中可以查找到文件名 `bin` 的目录项。这样也就知道了 `/usr/bin` 的 i 节点号，因而我们可以知道目录 `/usr/bin` 的目录所在位置，并在该目录中查找到 `vi` 文件的目录项。最终我们获得了文件路径名 `/usr/bin/vi` 的 i 节点号，从而可以从磁盘上得到该 i 节点号的 i 节点结构信息。

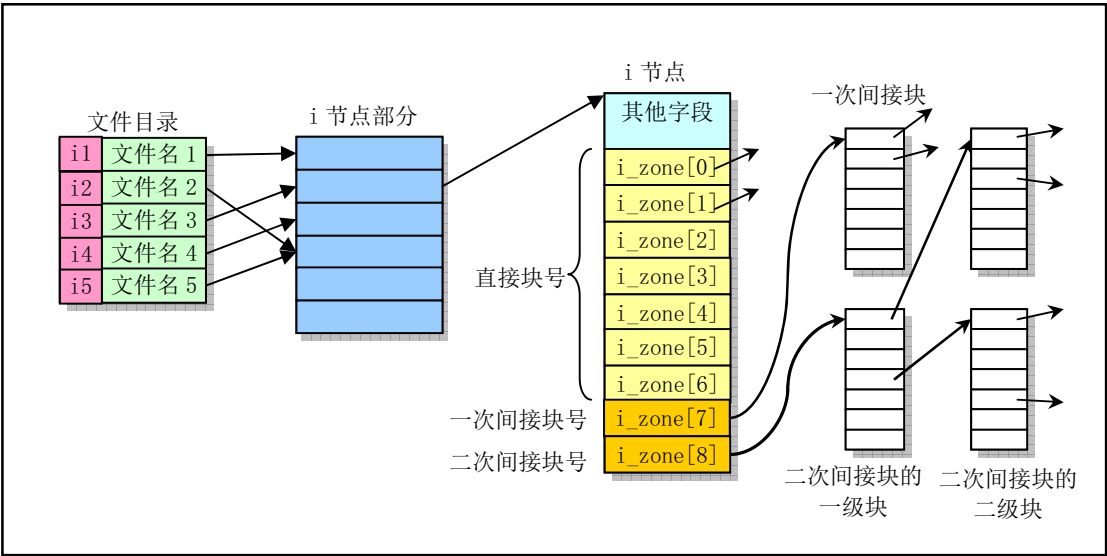


图 12-8 通过文件名最终找到对应文件磁盘块位置的示意图

如果从一个文件在磁盘上的分布来看，对于某个文件数据块信息的寻找过程可用图 12-9 表示（其中未画出引导块、超级块、i 节点和逻辑块位图）。

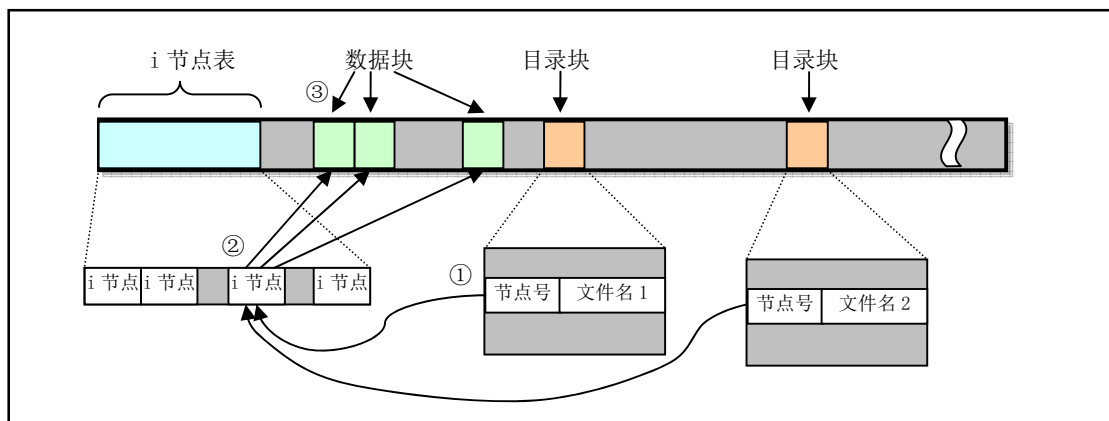


图 12-9 从文件名获取其数据块

通过对用户程序指定的文件名，我们可以找到对应目录项。根据目录项中的 i 节点号就可以找到 i 节点表中相应的 i 节点结构。 i 节点结构中包含着该文件数据的块号信息，因此最终可以得到文件名对应的数据信息。上图中有两个目录项指向了同一个 i 节点，因此根据这两个文件名都可以得到磁盘上相同的数据。每个 i 节点结构中都有一个链接计数字段 i_nlinks 记录着指向该 i 节点的目录项数，即文件的硬链接计数值。本例中该字段值为 2。在执行删除操作文件时，只有当 i 节点链接计数值等于 0 时内核才会真正删除磁盘上该文件的数据。另外，由于目录项中 i 节点号仅能用于当前文件系统中，因此不能使用一个文件系统的目录项来指向另一个文件系统中的 i 节点，即硬链接不能跨越文件系统。

与硬链接不同，符号链接类型的文件名目录项并不直接指向对应的 i 节点。符号链接目录项会在对应文件的数据块中存放某一文件的路径名字符串。当访问符号链接目录项时，内核就会读取该文件中的内容，然后根据其中的路径名字符串来访问指定的文件。因此符号链接可以不局限在一个文件系统中，我们可以在一个文件系统中建立一个指向另一个文件系统中文件名的符号链接。

在每个目录中还包含两个特殊的文件目录项，它们的名称分别固定是 '.' 和 '..'。 '.' 目录项中给出了当前目录的 i 节点号，而 '..' 目录项中给出了当前目录父目录的 i 节点号。因此在给出一个相对路径名时文件系统就可以利用这两个特殊目录项进行查找操作。例如要查找 `../kernel/Makefile`，就可以首先根据当前目录的 '..' 目录项得到父目录的 i 节点号，然后按照上面描述过程进行查找操作。

对于每个目录文件的目录项，其 i 节点中的链接计数字段值也表明连接到该目录的目录项数。因此每个目录文件的链接计数值起码为 2。其中一个包含目录文件的目录项链接，另一个是目录中 '..' 目录项的链接。例如我们在当前目录中建立一个名为 `mydir` 的子目录，那么在当前目录和该子目录中的链接示意图见图 12-10 所示。

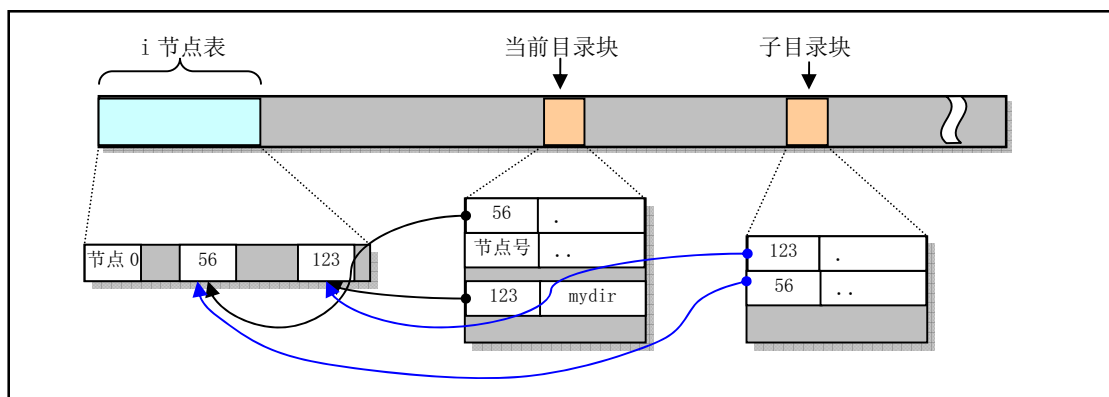


图 12-10 目录文件目录项和子目录链接

图中示出了我们在 i 节点号为 56 的目录中建立了一个 mydir 子目录，该子目录的 i 节点号是 123。在 mydir 子目录中的 '.' 目录项指向自己的 i 节点 123，而其 '..' 目录项则指向其父目录的 i 节点 56。可见，由于一个目录的目录项本身总是会有两个链接，若其中再包含子目录，那么父目录的 i 节点链接数就等于 2+子目录数。

12.1.2.3 目录结构例子

以 Linux 0.11 系统为例，我们来观察它的根目录项结构。在 bochs 中运行 Linux 0.11 系统之后，我们先列出其文件系统根目录项，包括其中隐含的 '.' 和 '..' 目录项。然后我们使用 hexdump 命令察看 '.' 或 '..' 文件的数据块内容，可以看出根目录包含的各个目录项内容。

```
[/usr/root]# cd /
[/]# ls -la
total 10
drwxr-xr-x 10 root    root      176 Mar 21  2004 .
drwxr-xr-x 10 root    4096      176 Mar 21  2004 ..
drwxr-xr-x  2 root    4096      912 Mar 21  2004 bin
drwxr-xr-x  2 root    root      336 Mar 21  2004 dev
drwxr-xr-x  2 root    root      224 Mar 21  2004 etc
drwxr-xr-x  8 root    root      128 Mar 21  2004 image
drwxr-xr-x  2 root    root        32 Mar 21  2004 mnt
drwxr-xr-x  2 root    root        64 Mar 21  2004 tmp
drwxr-xr-x 10 root    root      192 Mar 29  2004 usr
drwxr-xr-x  2 root    root        32 Mar 21  2004 var

[/]# hexdump .
00000000 0001 002e 0000 0000 0000 0000 0000 0000 // .
00000010 0001 2e2e 0000 0000 0000 0000 0000 0000 // ..
00000020 0002 6962 006e 0000 0000 0000 0000 0000 // bin
00000030 0003 6564 0076 0000 0000 0000 0000 0000 // dev
00000040 0004 7465 0063 0000 0000 0000 0000 0000 // etc
00000050 0005 7375 0072 0000 0000 0000 0000 0000 // usr
00000060 0115 6e6d 0074 0000 0000 0000 0000 0000 // mnt
00000070 0036 6d74 0070 0000 0000 0000 0000 0000 // tmp
00000080 0000 6962 2e6e 656e 0077 0000 0000 0000 // 空闲，未使用。
00000090 0052 6d69 6761 0065 0000 0000 0000 0000 // image
000000a0 007b 6176 0072 0000 0000 0000 0000 0000 // var
000000b0
[/]#
```

执行 'hexdump .' 命令后列出了 1 号 i 节点数据块中包含的所有目录项。每一行对应一个目录项，每行开始两字节是 i 节点号，随后的 14 字节是文件名或目录名字符串。若一个目录项中 i 节点号是 0，则表示该目录项没有被使用，或对应的文件已经被删除或移走。其中头两个目录项 ('.' 和 '..') 的 i 节点号均是 1 号。这是文件系统根目录结构的特殊之处，与其余子目录结构不同。

现在察看 etc/ 目录项。同样对 etc/ 目录使用 hexdump 命令，我们可以显示出 etc/ 子目录包含的目录项，见下面所示。

```
[/]# ls etc -la
total 32
```

```

drwxr-xr-x  2 root    root      224 Mar 21  2004 .
drwxr-xr-x 10 root    root      176 Mar 21  2004 ..
-rw-r--r--  1 root    root      137 Mar  4  2004 group
-rw-r--r--  1 root    root    11801 Mar  4  2004 magic
-rw-r--r--  1 root    root       11 Jan 22 18:12 mtab
-rw-r--r--  1 root    root     142 Mar  5  2004 mtools
-rw-r--r--  1 root    root     266 Mar  4  2004 passwd
-rw-r--r--  1 root    root     147 Mar  4  2004 profile
-rw-r--r--  1 root    root       57 Mar  4  2004 rc
-rw-r--r--  1 root    root    1034 Mar  4  2004 termcap
-rwx--x--x  1 root    root   10137 Jan 15 1992 update

[/]# hexdump etc
00000000 0004 002e 0000 0000 0000 0000 0000 0000 // .
00000010 0001 2e2e 0000 0000 0000 0000 0000 0000 // ..
00000020 0007 6372 0000 0000 0000 0000 0000 0000 // rc
00000030 000b 7075 6164 6574 0000 0000 0000 0000 // update
00000040 0113 6574 6d72 6163 0070 0000 0000 0000 // termcap
00000050 00ee 746d 6261 0000 0000 0000 0000 0000 // mtab
00000060 0000 746d 6261 007e 0000 0000 0000 0000 // 空闲, 未使用
00000070 007c 616d 6967 0063 0000 0000 0000 0000 // magic
00000080 0016 7270 666f 6c69 0065 0000 0000 0000 // profile
00000090 007e 6170 7373 6477 0000 0000 0000 0000 // passwd
00000a00 0081 7267 756f 0070 0000 0000 0000 0000 // group
00000b00 01ee 746d 6f6f 736c 0000 0000 0000 0000 // mtools
00000c00
[/]#
```

此时我们可以看出 `etc/` 目录名 `i` 节点对应的数据块中包含有该子目录下所有文件的目录项信息。其中目录项 `'.'` 的 `i` 节点正是 `etc/` 目录项自己的 `i` 节点号 4，而 `'..'` 的 `i` 节点是 `etc/` 父目录的 `i` 节点号 1。

12.1.3 高速缓冲区

高速缓冲区是文件系统访问块设备中数据的必经要道。为了访问文件系统等块设备上的数据，内核可以每次都访问块设备，进行读或写操作。但是每次 I/O 操作的时间与内存和 CPU 的处理速度相比是非常慢的。为了提高系统的性能，内核就在内存中开辟了一个高速数据缓冲区（池）（buffer cache），并将其划分成一个个与磁盘数据块大小相等的缓冲块来使用和管理，以期减少访问块设备的次数。在 linux 内核中，高速缓冲区位于内核代码和主内存区之间，参见图 2-5 所示。高速缓冲中存放着最近被使用过的各个块设备中的数据块。当需要从块设备中读取数据时，缓冲区管理程序首先会在高速缓冲中寻找。如果相应数据已经在缓冲中，就无需再从块设备上读。如果数据不在高速缓冲中，就发出读块设备的命令，将数据读到高速缓冲中。当需要把数据写到块设备中时，系统就会在高速缓冲区中申请一块空闲的缓冲块来临时存放这些数据。至于什么时候把数据真正地写到设备中去，则是通过设备数据同步实现的。

Linux 内核实现高速缓冲区的程序是 `buffer.c`。文件系统中其他程序通过指定需要访问的设备号和数据逻辑块号来调用它的块读写函数。这些接口函数有：块读取函数 `bread()`、块提前预读函数 `breada()` 和页块读取函数 `bread_page()`。页块读取函数一次读取一页内存所能容纳的缓冲块数（4 块）。

12.1.4 文件系统底层函数

文件系统的底层处理函数包含在以下 5 个文件中：

- `bitmap.c` 程序包括对 `i` 节点位图和逻辑块位图进行释放和占用处理函数。操作 `i` 节点位图的函数是 `free_inode()` 和 `new_inode()`，操作逻辑块位图的函数是 `free_block()` 和 `new_block()`。

- `truncate.c` 程序包括对数据文件长度截断为 0 的函数 `truncate()`。它将 `i` 节点指定的设备上文件长度截为 0，并释放文件数据占用的设备逻辑块。
 - `inode.c` 程序包括分配 `i` 节点函数 `iget()` 和放回对内存 `i` 节点存取函数 `iput()` 以及根据 `i` 节点信息取文件数据块在设备上对应的逻辑块号函数 `bmap()`。
 - `namei.c` 程序主要包括函数 `namei()`。该函数使用 `iget()`、`iput()` 和 `bmap()` 将给定的文件路径名映射到其 `i` 节点。
 - `super.c` 程序专门用于处理文件系统超级块，包括函数 `get_super()`、`put_super()` 和 `free_super()` 等。还包括几个文件系统加载/卸载处理函数和系统调用，如 `sys_mount()` 等。
- 这些文件中函数之间的层次关系如图 12-11 所示。

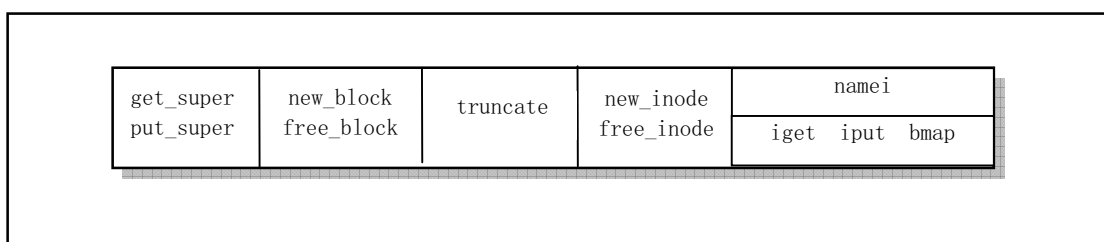


图 12-11 文件系统低层操作函数层次关系

12.1.5 文件中数据的访问操作

关于文件中数据的访问操作代码，主要涉及 5 个文件（见图 12-12 所示）：`block_dev.c`、`file_dev.c`、`char_dev.c`、`pipe.c` 和 `read_write.c`。前 4 个文件可以认为是块设备、字符设备、管道设备和普通文件与文件读写系统调用的接口程序，它们共同实现了 `read_write.c` 中的 `read()` 和 `write()` 系统调用。通过对被操作文件属性的判断，这两个系统调用会分别调用这些文件中的相关处理函数进行操作。

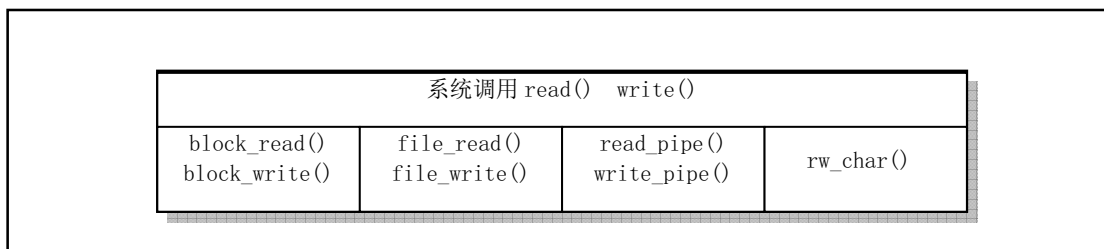


图 12-12 文件数据访问函数

`block_dev.c` 中的函数 `block_read()` 和 `block_write()` 是用于读写块设备特殊文件中的数据。所使用的参数指定了要访问的设备号、读写的起始位置和长度。

`file_dev.c` 中的 `file_read()` 和 `file_write()` 函数是用于访问一般的正规文件。通过指定文件对应的 `i` 节点和文件结构，从而可以知道文件所在的设备号和文件当前的读写指针。

`pipe.c` 文件中实现了管道读写函数 `read_pipe()` 和 `write_pipe()`。另外还实现了创建无名管道的系统调用 `pipe()`。管道主要用于在进程之间按照先进先出的方式传送数据，也可以用于使进程同步执行。有两种类型的管道：有名管道和无名管道。有名管道是使用文件系统的 `open` 调用建立的，而无名管道则使用系统调用 `pipe()` 来创建。在使用管道时，则都用正规文件的 `read()`、`write()` 和 `close()` 函数。只有发出 `pipe` 调用的后代，才能共享对无名管道的存取，而所有进程只要权限许可，都可以访问有名管道。

对于管道的读写，可以看成是一个进程从管道的一端写入数据，而另一个进程从管道的另一端读出

数据。内核存取管道中数据的方式与存取一般正规文件中数据的方式完全一样。为管道分配存储空间和为正规文件分配空间的不同之处是，管道只使用 *i* 节点的直接块。内核将 *i* 节点的直接块作为一个循环队列来管理，通过修改读写指针来保证先进先出的顺序。

对于字符设备文件，系统调用 `read()` 和 `write()` 会调用 `char_dev.c` 中的 `rw_char()` 函数来操作。字符设备包括控制台终端 (`tty`)、串口终端 (`ttyx`) 和内存字符设备。

另外，内核使用文件结构 `file`、文件表 `file_table[]` 和内存中的 *i* 节点表 `inode_table[]` 来管理对文件的操作访问。这些数据结构和表的定义可参见头文件 `include/linux/fs.h`。文件结构 `file` 被定义为如下所示。

```
struct file {
    unsigned short f_mode;           // 文件类型和访问属性 (RW 位)
    unsigned short f_flags;          // 文件打开和控制的标志。
    unsigned short f_count;          // 对应文件句柄引用计数。
    struct m_inode * f_inode;        // 指向对应内存 i 节点，即现在系统中的 v 节点。
    off_t f_pos;                     // 文件当前读写指针位置。
};
struct file file_table[NR_FILE]     // 文件表数组，共 64 项。
```

它用于在文件句柄与内存 *i* 节点表中 *i* 节点项之间建立关系。其中文件类型和访问属性字段 `f_mode` 与文件 *i* 节点结构中 `i_mode` 字段的含义相同，见前面的描述；`f_flags` 字段是打开文件调用函数 `open()` 中参数 `flag` 给出的一些打开操作控制标志的组合，这些标志定义在 `include/fcntl.h` 中。其中有以下一些标志：

```
// 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        // 如果文件不存在就创建。fcntl 函数不用。
12 #define O_EXCL        00200        // 独占使用文件标志。
13 #define O_NOCTTY      00400        // 不分配控制终端。
14 #define O_TRUNC       01000        // 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        // 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。
```

`file` 结构中的引用计数字段 `f_count` 指出本文件被文件句柄引用的次数计数；内存 *i* 节点结构字段 `f_inode` 指向本文件对应 *i* 节点表中的内存 *i* 节点结构项。文件表是内核中由文件结构项组成的数组，在 `linux 0.11` 内核中文件表最多可有 64 项，因此整个系统同时最多打开 64 个文件。在进程的数据结构（即进程控制块或称进程描述符）中，专门定义有本进程打开文件的文件结构指针数组 `filp[NR_OPEN]` 字段。其中 `NR_OPEN = 20`，因此每个进程最多可同时打开 20 个文件。该指针数组项的序号即对应文件的描述符值，而项的指针则指向文件表中打开的文件项。例如，`filp[0]` 即是进程当前打开文件描述符 0 对应的文件结构指针。

内核中 *i* 节点表 `inode_table[NR_INODE]` 是由内存 *i* 节点结构组成的数组，其中 `NR_INODE = 32`，因此在某一时刻内核中同时只能保存 32 个内存 *i* 节点信息。一个进程打开的文件和内核文件表以及相应内存 *i* 节点的关系可用图 12-13 来表示。图中一个文件被作为进程的标准输入打开（文件句柄 0），另一个被作为标准输出打开（文件句柄 1）。

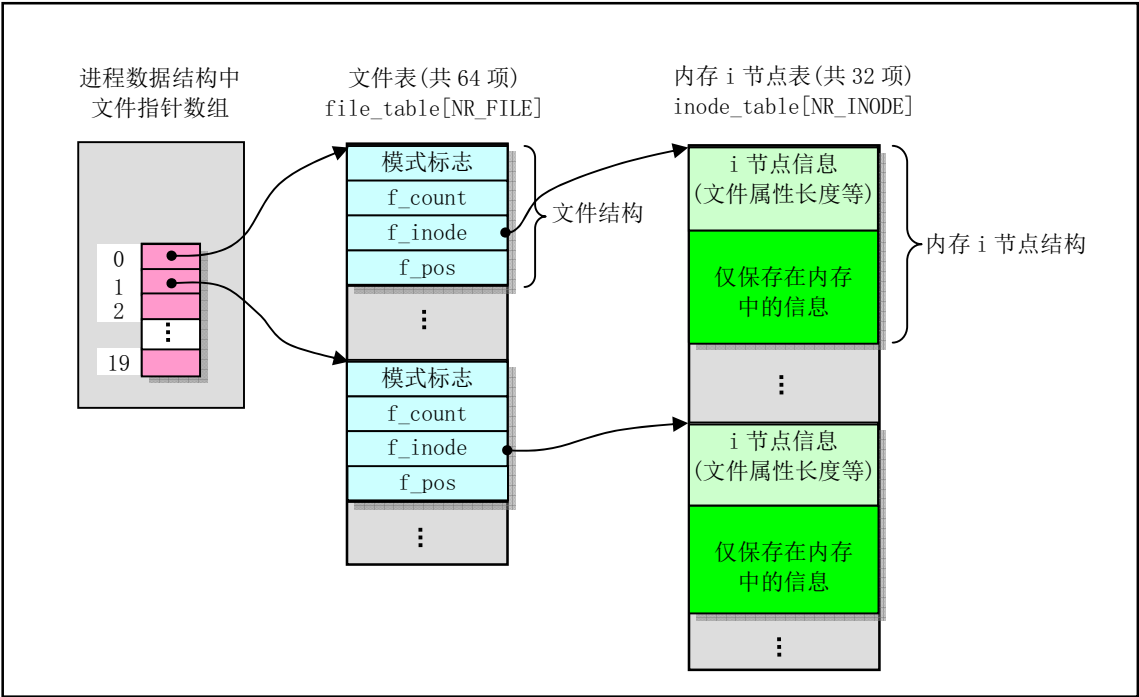


图 12-13 进程打开文件使用的内核数据结构

12.1.6 文件和目录管理系统调用

有关文件系统调用的上层实现，基本上包括图 12-14 中 5 个文件。

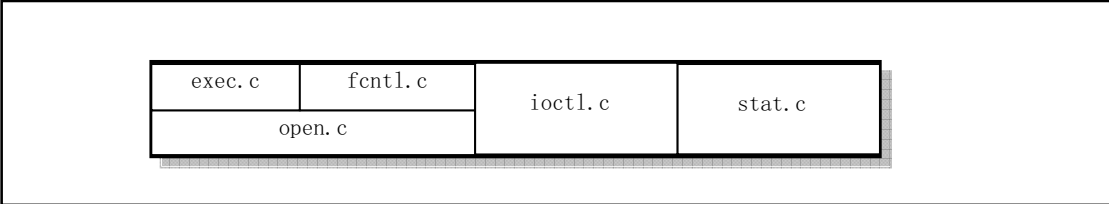


图 12-14 文件系统上层操作程序

open.c 文件用于实现与文件操作相关的系统调用。主要有文件的创建、打开和关闭，文件宿主和属性的修改、文件访问权限的修改、文件操作时间的修改和系统文件系统 **root** 的变动等。

exec.c 程序实现对二进制可执行文件和 **shell** 脚本文件的加载与执行。其中主要的函数是函数 **do_execve()**，它是系统中断调用(int 0x80)功能号 **__NR_execve()**调用的 C 处理函数，是 **exec()**函数簇的主要实现函数。

fcntl.c 实现了文件控制系统调用 **fcntl()**和两个文件句柄(描述符)复制系统调用 **dup()**和 **dup2()**。**dup2()**指定了新句柄的数值，而 **dup()**则返回当前值最小的未用句柄。句柄复制操作主要用在文件的标准输入/输出重定向和管道操作方面。

ioctl.c 文件实现了输入/输出控制系统调用 **ioctl()**。主要调用 **tty_ioctl()**函数，对终端的 I/O 进行控制。

stat.c 文件用于实现取文件状态信息系统调用 **stat()**和 **fstat()**。**stat()**是利用文件名取信息，而 **fstat()**是使用文件句柄(描述符)来取信息。

12.1.7 360KB 软盘中文件系统实例分析

为了加深对图 12-1 所示文件系统结构的理解，我们利用 Linux 0.11 系统在 360KB 规格软盘映像中建立一个 MINIX 1.0 文件系统，其中仅存放了一个名为 `hello.c` 的文件。

我们首先在 `bochs` 环境中运行下列命令来建立一个文件系统。

```
[/usr/root]# mkfs /dev/fd1 360           // 在第 2 个软驱中建立一个 360KB 的文件系统。
120 inodes                             // 它共有 120 个 i 节点，360 个盘块（逻辑块）。
360 blocks
Firstdatazone=8 (8)                   // 盘中数据区的开始盘块号是 8。
Zonesize=1024                          // 盘块大小为 1024 字节。
Maxsize=268966912                     // 最大文件长度（显然有误）。

[/usr/root]# mount /dev/fd1 /mnt        // 安装到/mnt 目录并复制一个文件到其中。
[/usr/root]# cp hello.c /mnt
[/usr/root]# ll -a /mnt                 // 有 3 个目录项。
total 3
drwxr-xr-x  2 root    root          48 Feb 23 17:48 .
drwxr-xr-x 10 root    root          176 Mar 21  2004 ..
-rw-----  1 root    root           74 Feb 23 17:48 hello.c

[/usr/root]# umount /dev/fd1           // 卸载该文件系统。
[/usr/root]#
```

对第 2 个软驱中的软盘（映像文件）执行 `mkfs` 命令后，会在盘上建立起一个 MINIX 文件系统。从命令执行后显示的内容可知，该文件系统共含有 120 个 i 节点、360 个盘块，盘中数据区起始的盘块号是 8，逻辑块的大小是 1024 字节，与盘块大小相同。并且可存放文件的最大长度为 268966912 字节（显然有误）。然后我们使用 `mount` 命令把这个含有 MINIX 文件系统的设备安装到目录 `/mnt` 上，并在往其中复制了一个文件 `hello.c` 后再卸载该文件系统。现在我们就已经制作好了一个只含有一个文件的 MINIX 文件系统。它被存放在 `bochs` 第 2 个软驱对应的磁盘映像文件（`diskb.img`）中。

现在我们来查看这个文件系统中的具体内容。为了方便，我们直接使用 Linux 0.11 系统中的 `hexdump` 命令来观察其中内容。你也可以退出 `bochs` 系统并使用 `UltraEdit` 等可修改二进制文件的编辑程序来查看。在对设备 `/dev/fd1` 执行 `hexdump` 命令后会显示以下内容（略作了整理）。

```
[/usr/root]# hexdump /dev/fd1 | more
00000000 44eb 4d90 6f74 6c6f 2073 0020 0102 0001 // 0x0000 - 0x03ff (1KB) 是引导块内容。
00000010 e002 4000 f00b 0009 0012 0002 0000 0000
00000020 0000 0000 0000 0000 0000 0000 0000 0000
*
00004000 0078 0168 0001 0001 0008 0000 1c00 1008 // 0x0400 - 0x07ff (1KB) 是超级块内容。
0000410 137f 0000 0000 0000 0000 0000 0000 0000
0000420 0000 0000 0000 0000 0000 0000 0000 0000
*
00008000 0007 0000 0000 0000 0000 0000 0000 ff00 // 0x0800 - 0x0bff (1KB) 是 i 节点位图内容。
0000810 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000c000 0007 0000 0000 0000 0000 0000 0000 0000 // 0x0c00 - 0x0fff (1KB) 是逻辑块位图内容。
0000c10 0000 0000 0000 0000 0000 0000 0000 0000
```

```

0000c20 0000 0000 0000 0000 0000 0000 0000 fffe ffff
0000c30 ffff ffff ffff ffff ffff ffff ffff ffff
*
0001000 41ed 0000 0030 0000 c200 421c 0200 0008 // 0x1000 - 0x1fff (4KB) 是 120 个 i 节点内容。
0001010 0000 0000 0000 0000 0000 0000 0000 0000
0001020 8180 0000 004a 0000 c200 421c 0100 0009
0001030 0000 0000 0000 0000 0000 0000 0000 0000
*
0002000 0001 002e 0000 0000 0000 0000 0000 0000 // 0x2000 - 0x23ff (1KB) 是 1 号根 i 节点数据。
0002010 0001 2e2e 0000 0000 0000 0000 0000 0000
0002020 0002 6568 6c6c 2e6f 0063 0000 0000 0000
0002030 0000 0000 0000 0000 0000 0000 0000 0000
*
0002400 6923 636e 756c 6564 3c20 7473 6964 2e6f // 0x2400 - 0x27ff (1KB) 是 hello.c 文件。
0002410 3e68 0a0a 6e69 2074 616d 6e69 2928 7b0a
0002420 090a 7270 6e69 6674 2228 6548 6c6c 2c6f
0002430 7720 726f 646c 5c21 226e 3b29 090a 6572
0002440 7574 6e72 3020 0a3b 0a7d 0000 0000 0000
0002450 0000 0000 0000 0000 0000 0000 0000 0000
--More--

```

现在我们逐一分析以上内容。根据图 12-1 我们知道，MINIX 1.0 文件系统的第 1 个盘块是一个引导盘块。因此盘块 0 (0x0000 - 0x03ff, 1KB) 是引导块内容。无论你的盘是否用来引导系统，每个新创建的文件系统都会保留一个引导盘块。对于新创建的磁盘映像文件，引导盘块应该全部为零。上述显示数据中引导盘块的内容是原来映像文件中遗留下来的数据，即 `mkfs` 命令在创建文件系统时不会修改引导盘块的内容。

盘块 1 (0x0400 - 0x07ff, 1KB) 是超级块内容。根据 MINIX 文件系统超级块数据结构 (参见图 12-3) 我们可以知道表 12-1 中所列的文件系统超级块信息，共有 18 个字节中包含有效内容。由于每逻辑块对 `ing` 的盘块数对数值为 0，因此，对于 MINIX 文件系统来说，其盘块大小就等于逻辑块 (区块) 大小。

表 12-1 360KB 磁盘中 MINIX 文件系统超级块信息

字段名称	超级块字段名称	内容
<code>s_ninodes</code>	i 节点数	0x0078 = 120 个
<code>s_nzones</code>	区块 (逻辑块) 数	0x0168 = 360 块
<code>s_imap_blocks</code>	i 节点位图所占块数	0x0001 块
<code>s_zmap_blocks</code>	区块位图所占块数	0x0001 块
<code>s_firstdatazone</code>	第一个数据块块号	0x0008
<code>s_log_zone_size</code>	Log2 (盘块数/区块)	0x0000
<code>s_max_size</code>	最大文件长度	0x10081c00 = 268966912 字节
<code>s_magic</code>	文件系统魔数	0x137f

盘块 2 (0x0800 - 0x0bff, 1KB) 包含 i 节点位图信息。由于该文件系统中总共有 120 个 i 节点，而每个比特位代表 1 个 i 节点结构，因此文件系统实际占用了该 1KB 大小盘块中的 $120/8 = 15$ 个字节，其中比特位值为 0 表示文件系统中相应 i 节点结构未被占用，1 表示已占用或保留。盘块中其余不用的字节比特位值均被 `mkfs` 命令初始化为 1。

从盘块 2 的数据中我们可以看出，第 1 个字节值是 0x07 (0b0000111)，即 i 节点位图的最开始的 3 个比特位已被占用。又前面说明可知，第 1 个比特位 (位 0) 保留不用。第 2 和第 3 个比特位分别说明

了文件系统的 1 号 i 节点和 2 号 i 节点已被使用，即后面 i 节点区中已经包含有 2 个 i 节点结构内容，实际上，1 号节点被用作文件系统的根 i 节点，2 号 i 节点被用于该文件系统上的唯一一个文件 `hello.c`，其中 i 节点结构的内容将在后面说明。

盘块 3 (0x0c00 - 0x0fff, 1KB) 是逻辑块位图内容。由于磁盘容量只有 360KB，因此文件系统实际使用了其中 360 个比特位，即 $360/8 = 45$ 个字节。由于逻辑块位图仅表示磁盘数据区中盘块被占用的情况，因此去除已被使用的功能块数 (1 引导块 + 1 超级块 + 1i 节点位图块 + 1 逻辑块位图块 + 4i 节点区盘块 = 8)，实际需要的比特位数是 $360 - 8 = 352$ 个比特位 (占用 44 字节)，再加上保留不能使用的位 0 比特位，共需要 353 个比特位。这也就是为什么最后一个 (第 45) 字节 (0xfe) 只有 1 个比特位是 0 的原因。

因此，当我们知道一个逻辑块在逻辑块位图中的比特位偏移值 `nr` 时，那么其对应的实际磁盘上盘块号 `block` 就等于 $nr + 8 - 1$ ，即 $block = nr + s_firstdatazone - 1$ 。而当我们想为一个磁盘上盘块号 `block` 求其在逻辑块位图中的比特位偏移值 (即数据区中块号) `nr` 时，则其为 $nr = block - s_firstdatazone + 1$ 。

与 i 节点位图类似，第 1 个字节的前 3 比特位也已经被占用。第 1 个 (位 0) 比特位不留不用，第 2 和第 3 个比特位说明磁盘数据区中已经被使用了 2 个盘块 (逻辑块)。实际上，位 1 代表的磁盘数据区中的第 1 个盘块被用于 1 号根 i 节点存放数据信息 (目录项)，位 2 代表的磁盘数据区中第 2 个盘块被用于 2 号节点保存相关数据信息。请注意，这里所说的数据信息是指 i 节点管理的数据内容，并非 i 节点结构的信息。i 节点本身的结构信息将保存在专门供存放 i 节点结构信息的 i 节点区中盘块内，即磁盘盘块 4--7。

盘块 4--7 (0x1000 - 0x1fff, 4KB) 4 个盘块专门用来存放 i 节点结构信息。因为文件系统供有 120 个 i 节点，而每个 i 节点占用 32 个字节 (参见图 12-4)，因此共需要 $120 \times 32 = 3840$ 字节，即需要占用 4 个盘块。由上面显示的数据我们可以看出，前 32 个字节已经保存了 1 号根 i 节点的内容，随后的 32 字节中保存了 2 号 i 节点的内容，见表 12-2 和表 12-3 所示。

表 12-2 1 号根 i 节点结构内容

字段名称	i 节点字段名称	值和说明
<code>i_mode</code>	文件的类型和属性	0x41ed (drwxr-xr-x)
<code>i_uid</code>	文件宿主用户 id	0x0000
<code>i_size</code>	文件长度	0x00000030 (48 字节)
<code>i_mtime</code>	修改时间	0x421cc200 (Feb 23 17:48)
<code>i_gid</code>	文件组 id	0x00
<code>i_nlinks</code>	链接数	0x02
<code>i_zone[9]</code>	文件所占用的逻辑块号数组	zone[0] = 0x0008, 其余项均为 0。

表 12-3 2 号 i 节点结构内容

字段名称	i 节点字段名称	值和说明
<code>i_mode</code>	文件的类型和属性	0x8180 (-rw-----)
<code>i_uid</code>	文件宿主用户 id	0x0000
<code>i_size</code>	文件长度	0x0000004a (74 字节)
<code>i_mtime</code>	修改时间	0x421cc200 (Feb 23 17:48)
<code>i_gid</code>	文件组 id	0x00
<code>i_nlinks</code>	链接数	0x01
<code>i_zone[9]</code>	文件所占用的逻辑块号数组	zone[0] = 0x0009, 其余项均为 0。

可以看出，1 号根 i 节点的数据块只有 1 块，其逻辑块号是 8，位于磁盘数据区中第 1 块上，长度是

30 字节。有前面小节可知一个目录项长度是 16 (0x10) 字节，因此这个逻辑块中共存有 3 个目录项 (0x30 字节)。因为是一个目录，所以其链接数是 2。

2 号 i 节点的数据块也同样只有 1 块，并位于磁盘数据区中第 2 块内，盘块号是 9。其中存有的数据长度是 74 字节，即是 `hello.c` 文件的字节长度。

盘块 8 (0x2000 - 0x23ff, 1KB) 就是 1 号根 i 节点的数据。其中存有 48 字节的 3 个目录项结构信息，见表 12-4 所示。

表 12-4 1 号根 i 节点的数据内容

项	节点号	文件名
1	0x0001	0x2e (.)
2	0x0001	0x2e,0x2e (..)
3	0x0002	0x68,0x65,0x6c,0x6c,0x6f,0x2e,0x63 (hello.c)

盘块 9 (0x2400 - 0x27ff, 1KB) 是 `hello.c` 文件内容。其中包含了 74 字节的文本信息。

12.2 Makefile 文件

12.2.1 功能描述

makefile 是文件系统子目录中程序编译的管理配置文件，供编译管理工具软件 `make` 使用。

12.2.2 代码注释

程序 12-1 linux/fs/Makefile

```

1 AR      =gar      # 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。
2 AS      =gas      # GNU 的汇编程序。
3 CC      =gcc      # GNU C 语言编译器。
4 LD      =gld      # GNU 的连接程序。

# C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
# -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
# 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-mstring-insns Linux 自己
# 填加的优化选项，以后不再使用；-nostdinc -I../include 不使用默认路径中的包含文件，而使
# 用这里指定目录中的(../include)。
5 CFLAGS  =-Wall -O -fstrength-reduce -fcombine-regs -fomit-frame-pointer \
6          -mstring-insns -nostdinc -I../include

# C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
# 出设备或指定的输出文件中；-nostdinc -I../include 同前。
7 CPP     =gcc -E -nostdinc -I../include
8

# 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
# 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止 (-S)，从而产生与
# 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
# 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。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；若目标是.o，而源文件是.s，则使用第 2 个规则；若目标是.o 文件而源文件

```

```

# 是 c 文件，则可直接使用第 3 个规则。
9 .c.s:
10     $(CC) $(CFLAGS) \
11     -S -o $*.s $<
# 将所有 *.c 文件编译成 *.o 目标文件。不进行连接。
12 .c.o:
13     $(CC) $(CFLAGS) \
14     -c -o $*.o $<
# 下面规则表示将所有 *.s 汇编程序文件编译成 *.o 目标文件。16 行是实现该操作的具体命令。
15 .s.o:
16     $(AS) -o $*.o $<
17
# 定义目标文件变量 OBJs。
18 OBJs= open.o read_write.o inode.o file_table.o buffer.o super.o \
19     block_dev.o char_dev.o file_dev.o stat.o exec.o pipe.o namei.o \
20     bitmap.o fcntl.o ioctl.o truncate.o
21
# 在有了先决条件 OBJs 后使用下面的命令连接成目标 fs.o。
# 选项 '-r' 用于指示生成可重定位的输出，即产生可以作为链接器 ld 输入的目标文件。
22 fs.o: $(OBJs)
23     $(LD) -r -o fs.o $(OBJs)
24
# 下面的规则用于清理工作。当执行 'make clean' 时，就会执行 26--27 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。
25 clean:
26     rm -f core *.o *.a tmp_make
27     for i in *.c;do rm -f `basename $$i .c`.s;done
28
# 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
# 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
# 文件中 '### Dependencies' 行后面的所有行（下面从 35 开始的行），并生成 tmp_make
# 临时文件（30 行的作用）。然后对 fs/目录下的每一个 C 文件执行 gcc 预处理操作。
# -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
# 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
# 文件名加上其依赖关系—该源文件中包含的所有头文件列表。把预处理结果都添加到临时
# 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
29 dep:
30     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
31     (for i in *.c;do $(CPP) -M $$i;done) >> tmp_make
32     cp tmp_make Makefile
33
34 ### Dependencies:
35 bitmap.o : bitmap.c ../include/string.h ../include/linux/sched.h \
36     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
37     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h
38 block_dev.o : block_dev.c ../include/errno.h ../include/linux/sched.h \
39     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
40     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
41     ../include/asm/segment.h ../include/asm/system.h
42 buffer.o : buffer.c ../include/stdarg.h ../include/linux/config.h \
43     ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
44     ../include/sys/types.h ../include/linux/mm.h ../include/signal.h \
45     ../include/linux/kernel.h ../include/asm/system.h ../include/asm/io.h

```

```
46 char_dev.o : char_dev.c ../include/errno.h ../include/sys/types.h \  
47 ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \  
48 ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \  
49 ../include/asm/segment.h ../include/asm/io.h  
50 exec.o : exec.c ../include/errno.h ../include/string.h \  
51 ../include/sys/stat.h ../include/sys/types.h ../include/a.out.h \  
52 ../include/linux/fs.h ../include/linux/sched.h ../include/linux/head.h \  
53 ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \  
54 ../include/asm/segment.h  
55 fcntl.o : fcntl.c ../include/string.h ../include/errno.h \  
56 ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \  
57 ../include/sys/types.h ../include/linux/mm.h ../include/signal.h \  
58 ../include/linux/kernel.h ../include/asm/segment.h ../include/fcntl.h \  
59 ../include/sys/stat.h  
60 file_dev.o : file_dev.c ../include/errno.h ../include/fcntl.h \  
61 ../include/sys/types.h ../include/linux/sched.h ../include/linux/head.h \  
62 ../include/linux/fs.h ../include/linux/mm.h ../include/signal.h \  
63 ../include/linux/kernel.h ../include/asm/segment.h  
64 file_table.o : file_table.c ../include/linux/fs.h ../include/sys/types.h  
65 inode.o : inode.c ../include/string.h ../include/sys/stat.h \  
66 ../include/sys/types.h ../include/linux/sched.h ../include/linux/head.h \  
67 ../include/linux/fs.h ../include/linux/mm.h ../include/signal.h \  
68 ../include/linux/kernel.h ../include/asm/system.h  
69 ioctl.o : ioctl.c ../include/string.h ../include/errno.h \  
70 ../include/sys/stat.h ../include/sys/types.h ../include/linux/sched.h \  
71 ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \  
72 ../include/signal.h  
73 namei.o : namei.c ../include/linux/sched.h ../include/linux/head.h \  
74 ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \  
75 ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h \  
76 ../include/string.h ../include/fcntl.h ../include/errno.h \  
77 ../include/const.h ../include/sys/stat.h  
78 open.o : open.c ../include/string.h ../include/errno.h ../include/fcntl.h \  
79 ../include/sys/types.h ../include/utime.h ../include/sys/stat.h \  
80 ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \  
81 ../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \  
82 ../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h  
83 pipe.o : pipe.c ../include/signal.h ../include/sys/types.h \  
84 ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \  
85 ../include/linux/mm.h ../include/asm/segment.h  
86 read_write.o : read_write.c ../include/sys/stat.h ../include/sys/types.h \  
87 ../include/errno.h ../include/linux/kernel.h ../include/linux/sched.h \  
88 ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \  
89 ../include/signal.h ../include/asm/segment.h  
90 stat.o : stat.c ../include/errno.h ../include/sys/stat.h \  
91 ../include/sys/types.h ../include/linux/fs.h ../include/linux/sched.h \  
92 ../include/linux/head.h ../include/linux/mm.h ../include/signal.h \  
93 ../include/linux/kernel.h ../include/asm/segment.h  
94 super.o : super.c ../include/linux/config.h ../include/linux/sched.h \  
95 ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \  
96 ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \  
97 ../include/asm/system.h ../include/errno.h ../include/sys/stat.h  
98 truncate.o : truncate.c ../include/linux/sched.h ../include/linux/head.h \  

```



```

99  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
100  ../include/signal.h ../include/sys/stat.h

```

12.3 buffer.c 程序

从本节起，我们对 fs/目录下的程序逐一进行说明和注释。按照本章第 2 节中的描述，本章的程序可以被划分成 4 个部分：①高速缓冲管理；②文件底层操作；③文件数据访问；④文件高层访问控制。这里首先对第 1 部分的高速缓冲管理程序进行描述。这部分仅包含一个程序 buffer.c。

12.3.1 功能描述

buffer.c 程序用于对高速缓冲区(池)进行操作和管理。高速缓冲区位于内核代码块和主内存区之间，见图 12-15 中所示。高速缓冲区在块设备与内核其他程序之间起着一个桥梁作用。除了块设备驱动程序以外，内核程序如果需要访问块设备中的数据，就都需要经过高速缓冲区来间接地操作。

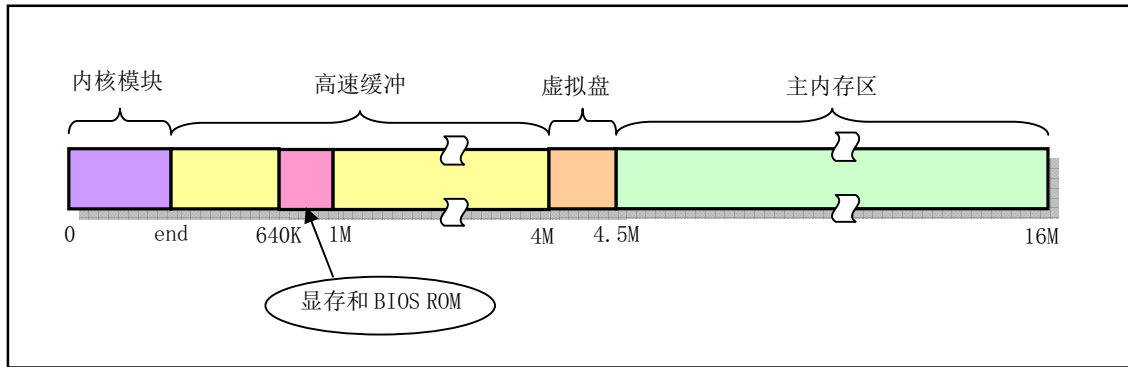


图 12-15 高速缓冲区在整个物理内存中所处的位置

图中高速缓冲区的起始位置从内核模块末段 end 标号开始，end 是内核模块链接期间由链接程序 ld 设置的一个外部变量，内核代码中没有定义这个符号。当在连接生成 system 模块时，ld 程序设置了 end 的地址，它等于 data_start + datasize + bss_size，即 bss 段结束后的第 1 个有效地址，也即内核模块的末端。另外，链接器还设置了 etext 和 edata 两个外部变量。它们分别表示代码段后第 1 个地址和数据段后第 1 个地址。

整个高速缓冲区被划分成 1024 字节大小的缓冲块，正好与块设备上的磁盘逻辑块大小相同。高速缓冲采用 hash 表和包含所有缓冲块的链表进行操作管理。在缓冲区初始化过程中，初始化程序从整个缓冲区的两端开始，分别同时设置缓冲块头结构和划分出对应的缓冲块，见图 12-16 所示。缓冲区的高端被划分成一个个 1024 字节的缓冲块，低端则分别建立起对应各缓冲块的缓冲头结构 buffer_head (include/linux/fs.h, 68 行)。该头结构用于描述对应缓冲块的属性，并且用于把所有缓冲头连接成链表。直到它们之间已经不能再划分出缓冲块为止。

所有缓冲块的 buffer_head 被链接成一个双向链表结构，见图 12-17 所示。图中 free_list 指针是该链表的头指针，指向空闲块链表中第一个“最为空闲的”缓冲块，即近期最少使用的缓冲块。而该缓冲块的反向指针 b_prev_free 则指向缓冲块链表中最后一个缓冲块，即最近刚使用的缓冲块。缓冲块的缓冲头数据结构为：

```

struct buffer_head {

```

```

char * b_data;                // 指向该缓冲块中数据区 (1024 字节) 的指针。
unsigned long b_blocknr;      // 块号。
unsigned short b_dev;         // 数据源的设备号 (0 = free)。
unsigned char b_uptodate;     // 更新标志: 表示数据是否已更新。
unsigned char b_dirt;         // 修改标志: 0- 未修改(clean), 1- 已修改(dirty)。
unsigned char b_count;        // 使用该块的用户数。
unsigned char b_lock;         // 缓冲块是否被锁定。0- ok, 1- locked
struct task_struct * b_wait;  // 指向等待该缓冲块解锁的任务。
struct buffer_head * b_prev;  // hash 队列上一块 (这四个指针用于缓冲块管理)。
struct buffer_head * b_next;  // hash 队列下一块。
struct buffer_head * b_prev_free; // 空闲表上一块。
struct buffer_head * b_next_free; // 空闲表下一块。
};

```

其中字段 `b_lock` 是锁定标志, 表示驱动程序正在对该缓冲块内容进行修改, 因此该缓冲块处于忙状态而正被锁定。该标志与缓冲块的其他标志无关, 主要用于 `blk_drv/ll_rw_block.c` 程序中在更新缓冲块中数据信息时锁定缓冲块。因为在更新缓冲块中数据时, 当前进程会自愿去睡眠等待, 从而别的进程就有机会访问该缓冲块。因此, 此时为了不让其他进程使用其中的数据就一定要在睡眠之前锁定缓冲块。

字段 `b_count` 是缓冲管理程序 `buffer` 使用的计数值, 表示相应缓冲块正被各个进程使用 (引用) 的次数, 因此这个字段用于对缓冲块的程序引用计数管理, 也与缓冲块的其他标志无关。当引用计数不为 0 时, 缓冲管理程序就不能释放相应缓冲块。空闲块即是 `b_count = 0` 的块。当 `b_count = 0` 时, 表示相应缓冲块未被使用 (free), 否则则表示它正在被使用着。对于程序申请的缓冲块, 若缓冲管理程序能够从 hash 表中得到已存在的指定块时, 就会将该块的 `b_count` 增 1 (`b_count++`)。若缓冲块是重新申请得到的未被使用的块, 则其头结构中的 `b_count` 被设置为等于 1。当程序释放其对一个块的引用时, 该块的引用次数就会相应地递减 (`b_count--`)。由于标志 `b_lock` 表示其他程序正在使用并锁定了指定的缓冲块, 因此对于 `b_lock` 置位的缓冲块来讲, 其 `b_count` 肯定大于 0。

字段 `b_dirt` 是脏标志, 说明缓冲块中内容是否已被修改而与块设备上的对应数据块内容不同 (延迟写)。字段 `b_uptodate` 是数据更新 (有效) 标志, 说明缓冲块中数据是否有效。初始化或释放块时这两个标志均设置成 0, 表示该缓冲块此时无效。当数据被写入缓冲块但还没有被写入设备时则 `b_dirt = 1`, `b_uptodate = 0`。当数据被写入块设备或刚从块设备中读入缓冲块则数据变成有效, 即 `b_uptodate = 1`。请注意有一种特殊情况。即在新申请一个设备缓冲块时 `b_dirt` 与 `b_uptodate` 都为 1, 表示缓冲块中数据虽然与块设备上的不同, 但是数据仍然是有效的 (更新的)。

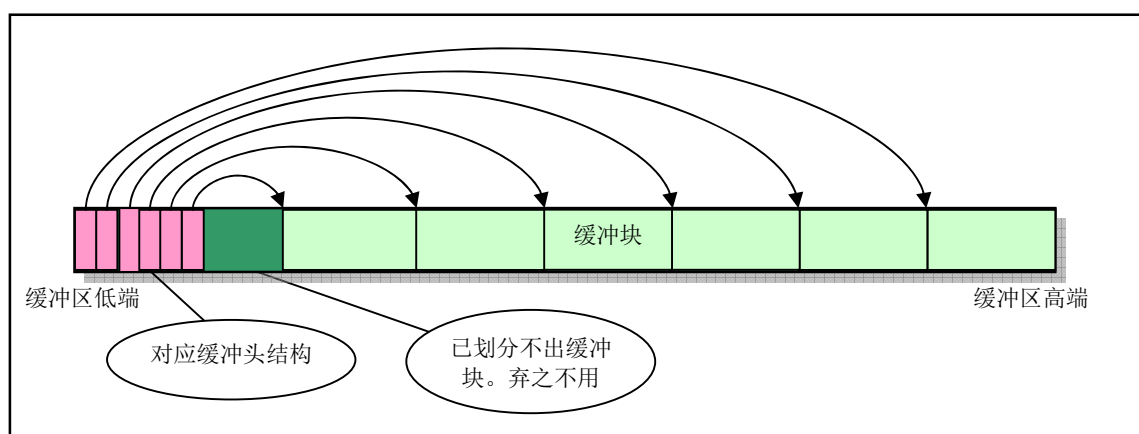


图 12-16 高速缓冲区的初始化

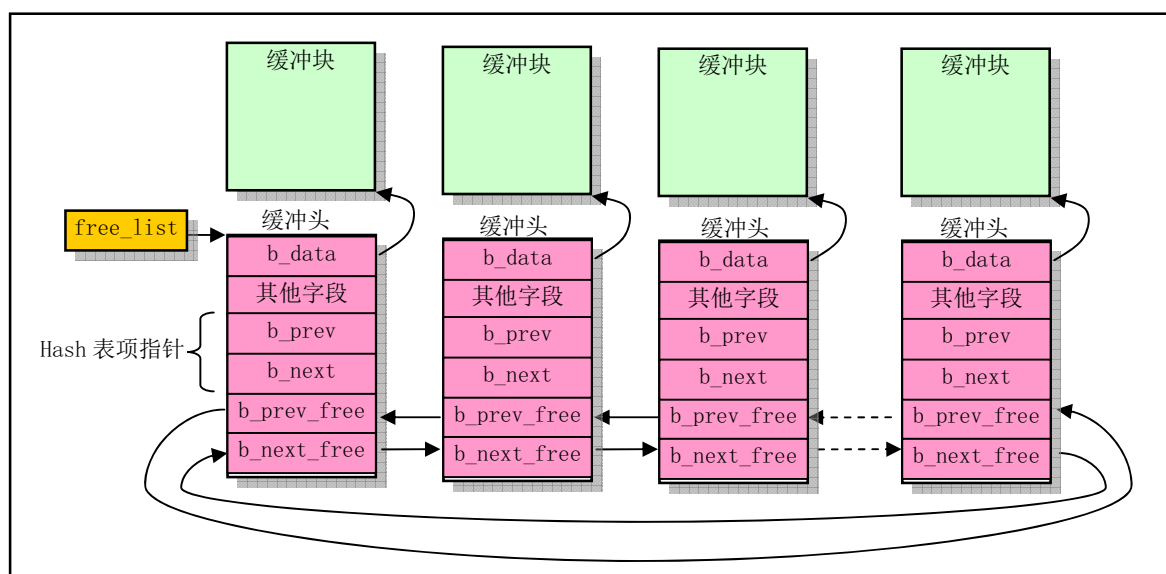


图 12-17 所有缓冲块组成的双向循环链表结构

图中缓冲头结构中“其他字段”包括块设备号、缓冲数据的逻辑块号，这两个字段唯一确定了缓冲块中数据对应的块设备和数据块。另外还有几个状态标志：数据有效（更新）标志、修改标志、数据被使用的进程数和本缓冲块是否上锁标志。

内核程序在使用高速缓冲区中的缓冲块时，是指定设备号（dev）和所要访问设备数据的逻辑块号（block），通过调用缓冲块读取函数 `bread()`、`bread_page()` 或 `breada()` 进行操作。这几个函数都使用缓冲区搜索管理函数 `getblk()`，用于在所有缓冲块中寻找最为空闲的缓冲块。该函数将在下面重点说明。在系统释放缓冲块时，需要调用 `brelse()` 函数。所有这些缓冲块数据存取和管理函数的调用层次关系可用图 12-18 来描述。

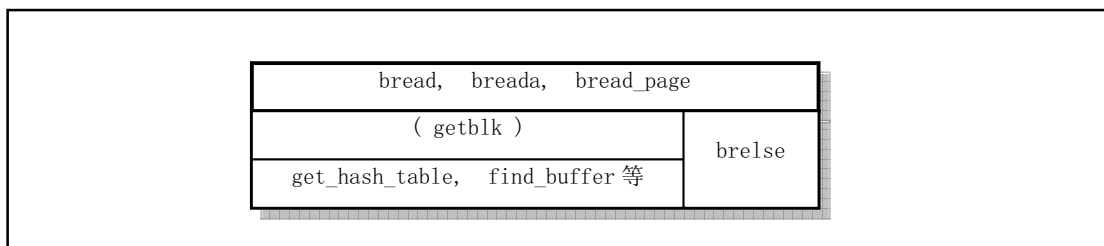


图 12-18 缓冲区管理函数之间的层次关系

为了能够快速而有效地在缓冲区中寻找判断出请求的数据块是否已经被读入到缓冲区中，`buffer.c` 程序使用了具有 307 个 `buffer_head` 指针项的 hash（散列）数组表结构。Hash 表所使用的散列函数由设备号和逻辑块号通过异或操作组合而成。程序中使用的具体 hash 函数是：(设备号^逻辑块号) Mod 307。图 12-17 中指针 `b_prev`、`b_next` 就是用于 hash 表中散列在同一项上多个缓冲块之间的双向链接，即把 hash 函数计算出的具有相同散列值的缓冲块链接在散列数组同一项链表上。有关散列队列上缓冲块的操作方式，可参见《Unix 操作系统设计》一书第 3 章中的详细描述。对于动态变化的 hash 表结构某一时刻的状态可参见图 12-19 所示。

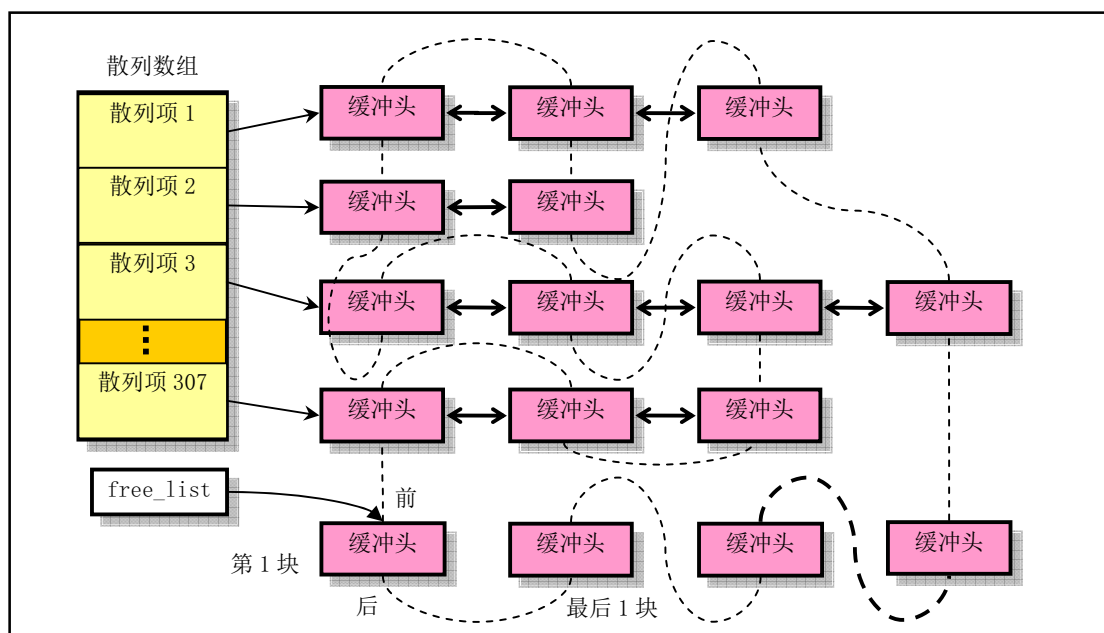


图 12-19 某一时刻内核中缓冲块散列队列示意图

其中，双箭头横线表示散列在同一 hash 表项中缓冲块头结构之间的双向链接指针。虚线表示缓冲区中所有缓冲块组成的一个双向循环链表（即所谓空闲链表），而 free_list 是该链表最为空闲的缓冲块处的头指针。实际上这个双向链表是一个最近最少使用 LRU（Least Recently Used）链表。下面我们对缓冲块搜索函数 getblk() 进行详细说明。

上面提及的三个函数在执行时都调用了 getblk()，以获取适合的空闲缓冲块。该函数首先调用 get_hash_table() 函数，在 hash 表队列中搜索指定设备号和逻辑块号的缓冲块是否存在。如果存在就立刻返回对应缓冲头结构的指针；如果不存在，则从空闲链表头开始，对空闲链表进行扫描，寻找一个空闲缓冲块。在寻找过程中还要对找到的空闲缓冲块作比较，根据赋予修改标志和锁定标志组合而成的权值，比较哪个空闲块最适合。若找到的空闲块既没有被修改也没有被锁定，就不用继续寻找了。若没有找到空闲块，则让当前进程进入睡眠状态，待继续执行时再次寻找。若该空闲块被锁定，则进程也需进入睡眠，等待其他进程解锁。若在睡眠等待的过程中，该缓冲块又被其他进程占用，那么只要再重新开始搜索缓冲块。否则判断该缓冲块是否已被修改过，若是，则将该块写盘，并等待该块解锁。此时如果该缓冲块又被别的进程占用，那么又一次全功尽弃，只好再重头开始执行 getblk()。在经历了以上折腾后，此时有可能出现另外一个意外情况，也就是在我们睡眠时，可能其他进程已经将我们所需要的缓冲块加进了 hash 队列中，因此这里需要最后一次搜索一下 hash 队列。如果真的在 hash 队列中找到了我们所需要的缓冲块，那么我们又得对找到的缓冲块进行以上判断处理，因此，又一次需要重头开始执行 getblk()。最后，我们才算找到了一块没有被进程使用、没有被上锁，而且是干净（修改标志未置位）的空闲缓冲块。于是我们就将该块的引用次数置 1，并复位其他几个标志，然后从空闲表中移出该块的缓冲头结构。在设置了该缓冲块所属的设备号和相应的逻辑号后，再将其插入 hash 表对应表项首部并链接到空闲队列的末尾处。由于搜索空闲块是从空闲队列头开始的，因此这种先从空闲队列中移出并使用最近不常用的缓冲块，然后再重新插入到空闲队列尾部的操作也就实现了最近最少使用 LRU 算法。最终，返回该缓冲块头的指针。整个 getblk() 处理过程可参见图 12-20 所示。

从上述分析可以可知，函数在每次获取新的空闲缓冲块时，就会把它移到 free_list 头指针所指链表的最后面，即越靠近链表末端的缓冲块被使用的时间就越近。因此如果 hash 表中没有找到对应缓冲块，就会在搜索新空闲缓冲块时从 free_list 链表头处开始搜索。可以看出，内核取得缓冲块的算法使用了以下策略：

- 如果指定的缓冲块存在于 hash 表中，则说明已经得到可用缓冲块，于是直接返回；
- 否则就需要在链表中从 free_list 头指针处开始搜索，即从最近最少使用的缓冲块处开始。

因此最理想的情况是找到一个完全空闲的缓冲块，即 b_dirt 和 b_lock 标志均为 0 的缓冲块；但是如果不能满足这两个条件，那么就需要根据 b_dirt 和 b_lock 标志计算出一个值。因为设备操作通常很耗时，所以在计算时需加大 b_dirt 的权重。然后我们在计算结果值最小的缓冲块上等待（如果缓冲块已经上锁）。最后当标志 b_lock 为 0 时，表示所等待的缓冲块原内容已经写到块设备上。于是 getblk() 就获得了一块空闲缓冲块。

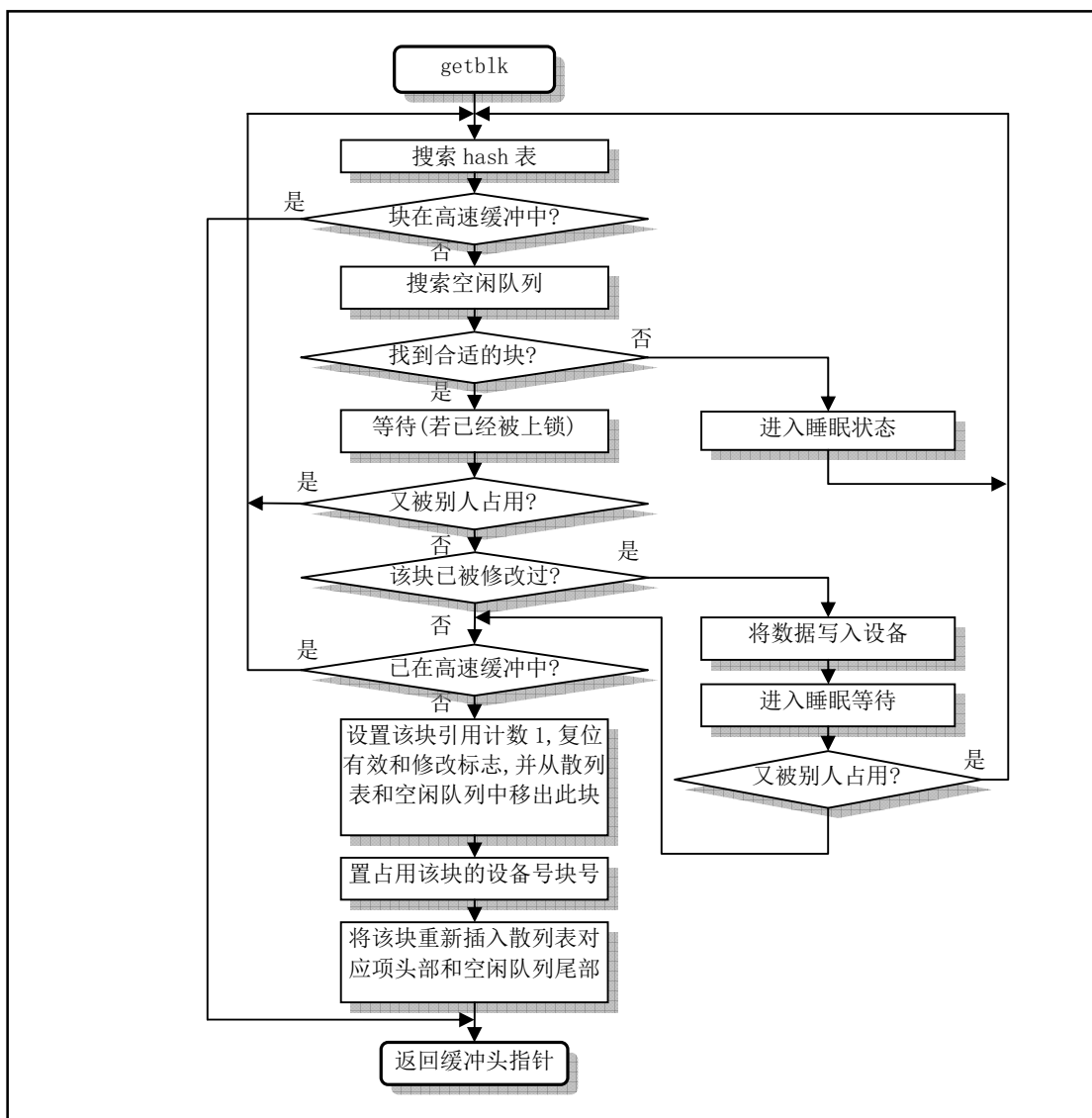


图 12-20 getblk() 函数执行流程图

由以上处理我们可以看到，getblk() 返回的缓冲块可能是一个新的空闲块，也可能正好是含有我们需要数据的缓冲块，它已经存在于高速缓冲区中。因此对于读取数据块操作（bread()），此时就要判断该缓冲块的更新标志，看看所含数据是否有效，如果有效就可以直接将该数据块返回给申请的程序。否则就需要调用设备的低层块读写函数（ll_rw_block()），并同时让自己进入睡眠状态，等待数据被读入缓冲块。在醒来后再判断数据是否有效了。如果有效，就可将此数据返给申请的程序，否则说明对设备的读操作

失败了，没有取到数据。于是，释放该缓冲块，并返回 NULL 值。图 12-21 是 `bread()` 函数的框图。`breada()` 和 `bread_page()` 函数与 `bread()` 函数类似。

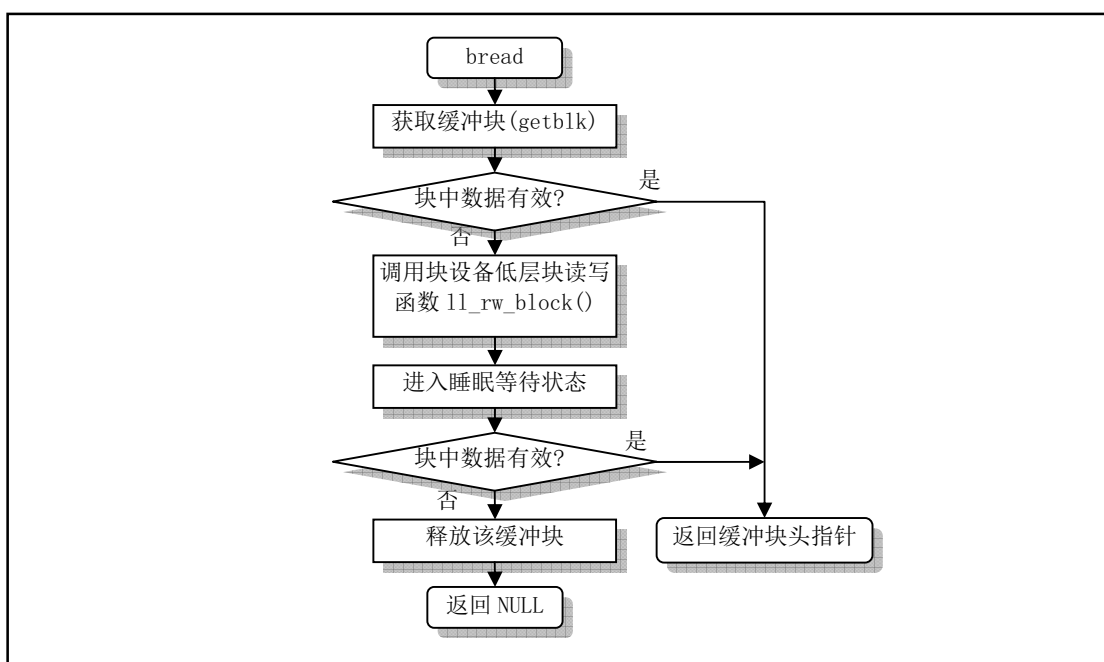


图 12-21 `bread()` 函数执行流程框图

当程序不再需要使用一个缓冲块中的数据时，就调用 `brelse()` 函数，释放该缓冲块并唤醒因等待该缓冲块而进入睡眠状态的进程。注意，空闲缓冲块链表中的缓冲块，并不是都是空闲的。只有当被写盘刷新、解锁且没有其他进程引用时（引用计数=0），才能挪作它用。

综上所述，高速缓冲区在提高对块设备的访问效率和增加数据共享方面起着重要的作用。除驱动程序以外，内核其他上层程序对块设备的读写操作需要经过高速缓冲区管理程序来间接地实现。它们之间的主要联系是通过高速缓冲区管理程序中的 `bread()` 函数和块设备低层接口函数 `ll_rw_block()` 来实现。上层程序若要访问块设备数据就通过 `bread()` 向缓冲区管理程序申请。如果所需的数据已经在高速缓冲区中，管理程序就会将数据直接返回给程序。如果所需的数据暂时还不在于缓冲区中，则管理程序会通过 `ll_rw_block()` 向块设备驱动程序申请，同时让程序对应的进程睡眠等待。等到块设备驱动程序把指定的数据放入高速缓冲区后，管理程序才会返回给上层程序。见图 12-22 所示。

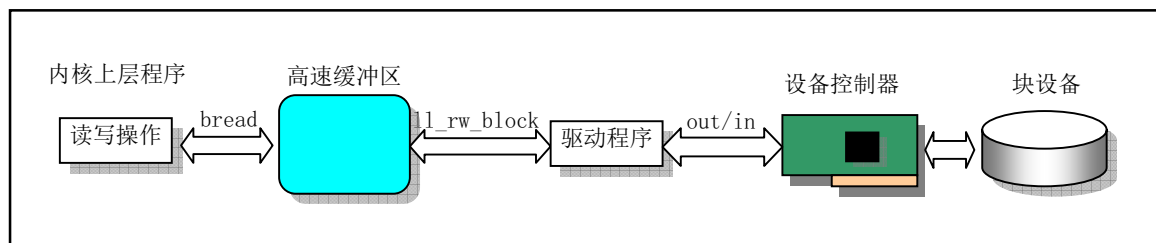


图 12-22 内核程序块设备访问操作

对于更新和同步（Synchronization）操作，其主要作用是让内存中的一些缓冲块内容与磁盘等块设备上的信息一致。`sync_inodes()` 的主要作用是把 `i` 节点表 `inode_table` 中的 `i` 节点信息与磁盘上的一致起来。但需要经过系统高速缓冲区这一中间环节。实际上，任何同步操作都被分成了两个阶段：

1. 数据结构信息与高速缓冲区中的缓冲块同步问题，由驱动程序独立负责；
2. 高速缓冲区中数据块与磁盘对应块的同步问题，由这里的缓冲管理程序负责。

`sync_inodes()`函数不会直接与磁盘打交道，它只能前进到缓冲区这一步，即只负责与缓冲区中的信息同步。剩下的需要缓冲管理程序负责。为了让 `sync_inodes()`知道哪些 `i` 节点与磁盘上的不同，就必须首先让缓冲区中内容与磁盘上的内容一致。这样 `sync_inodes()`通过与当前磁盘在缓冲区中的最新数据比较才能知道哪些磁盘 `inode` 需要修改和更新。最后再进行第二次高速缓冲区与磁盘设备的同步操作，做到内存中的数据与块设备中的数据真正的同步。

12.3.2 代码注释

程序 12-2 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 */
14 /*
15 *  '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
16 *  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
17 *  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。但需要非常快地
18 *  （我希望是这样）。
19 */
20
21 /*
22 *  NOTE! There is one discordant note here: checking floppies for
23 *  disk change. This is where it fits best, I think, as it should
24 *  invalidate changed floppy-disk-caches.
25 */
26 /*
27 *  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
28 *  该程序最好的地方了，因为它需要使已更换软盘缓冲失效。
29 */
30
31 #include <stdarg.h>          // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
32                             // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
33                             // vsprintf、vprintf、vfprintf 函数。
34
35 #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。
36 #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，
37                             // 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。
38 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
39 #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。
40 #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，参见图 12-15。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内含有的缓冲块个数。这里，NR_BUFFERS 是一个定义在 linux/fs.h 头
// 文件第 34 行的宏，其值即是变量名 nr_buffers，并且在 fs.h 文件第 166 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”。它将在后面的缓冲区初始化函数
// 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
64     // 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。
65     // 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，
66     // 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
67     // 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
68     // 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
69     // 指定设备的缓冲块，
70     bh = start_buffer; // bh 指向缓冲区开始处。
71     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
72         if (bh->b_dev != dev) // 不是设备 dev 的缓冲块则继续。
73             continue;
74         wait_on_buffer(bh); // 等待缓冲区解锁（如果已上锁的话）。
75         if (bh->b_dev == dev && bh->b_dirt)
76             ll_rw_block(WRITE, bh);
77     }
78     // 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。
79     sync_inodes();
80     // 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
81     // 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
82     // 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
83     // 脏的缓冲块与设备中数据同步。
84     bh = start_buffer;
85     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
86         if (bh->b_dev != dev)
87             continue;
88         wait_on_buffer(bh);
89         if (bh->b_dev == dev && bh->b_dirt)
90             ll_rw_block(WRITE, bh);
91     }
92     return 0;
93 }
94
95 // 使指定设备在高速缓冲区中的数据无效。
96 // 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。
97 void inline invalidate_buffers(int dev)
98 {
99     int i;
100    struct buffer_head * bh;
101
102    bh = start_buffer;
103    for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
104        if (bh->b_dev != dev) // 如果不是指定设备的缓冲块，则
105            continue; // 继续扫描下一块。
106        wait_on_buffer(bh); // 等待该缓冲区解锁（如果已被上锁）。
107    }

```

```

// 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
117     // 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
118     // 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。
119     if (MAJOR(dev) != 2)
120         return;
121     if (!floppy change(dev & 0x03))
122         return;
123     // 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
124     // 设备的 i 节点和数据块信息所占据的高速缓冲块无效。
125     for (i=0 ; i<NR_SUPER ; i++)
126         if (super block[i].s_dev == dev)
127             put_super(super block[i].s_dev);
128     invalidate inodes(dev);
129     invalidate buffers(dev);
130 }
131
132 // 下面两行代码是 hash（散列）函数定义和 hash 表项的计算宏。
133 // 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
134 // 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
135 // 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 0.11 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)
129 #define _hash(dev,block) _hash_table[_hashfn(dev,block)]
130
// 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。
131 static inline void _remove_from_queues(struct _buffer_head * bh)
132 {
133     /* remove from hash-queue */
134     /* 从 hash 队列中移除缓冲块 */
135     if (bh->b_next)
136         bh->b_next->b_prev = bh->b_prev;
137     if (bh->b_prev)
138         bh->b_prev->b_next = bh->b_next;
139     // 如果该缓冲块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。
140     if (_hash(bh->b_dev, bh->b_blocknr) == bh)
141         _hash(bh->b_dev, bh->b_blocknr) = bh->b_next;
142     /* remove from free list */
143     /* 从空闲缓冲块表中移除缓冲块 */
144     if (!(bh->b_prev_free) || !(bh->b_next_free))
145         panic("Free block list corrupted");
146     bh->b_prev_free->b_next_free = bh->b_next_free;
147     bh->b_next_free->b_prev_free = bh->b_prev_free;
148     // 如果空闲链表头指向本缓冲区，则让其指向下一缓冲区。
149     if (free_list == bh)
150         free_list = bh->b_next_free;
151 }
152
// 将缓冲块插入空闲链表尾部，同时放入 hash 队列中。
153 static inline void _insert_into_queues(struct _buffer_head * bh)
154 {
155     /* put at end of free list */
156     /* 放在空闲链表末尾处 */
157     bh->b_next_free = free_list;
158     bh->b_prev_free = free_list->b_prev_free;
159     free_list->b_prev_free->b_next_free = bh;
160     free_list->b_prev_free = bh;
161     /* put the buffer in new hash-queue if it has a device */
162     /* 如果该缓冲块对应一个设备，则将其插入新 hash 队列中 */
163     // 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
164     // 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
165     // 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
166     // 才被纠正。
167     bh->b_prev = NULL;
168     bh->b_next = NULL;
169     if (!bh->b_dev)
170         return;
171     bh->b_next = _hash(bh->b_dev, bh->b_blocknr);
172     _hash(bh->b_dev, bh->b_blocknr) = bh;
173     bh->b_next->b_prev = bh; // 此句前应添加 “if (bh->b_next)” 判断。

```

```

164 }
165
166 // 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。
167 // 如果找到则返回缓冲区块的指针，否则返回 NULL。
168 static struct buffer\_head * find\_buffer(int dev, int block)
169 {
170     struct buffer\_head * tmp;
171
172     // 搜索 hash 表，寻找指定设备号和块号的缓冲区块。
173     for (tmp = hash(dev, block) ; tmp != NULL ; tmp = tmp->b_next)
174         if (tmp->b_dev==dev && tmp->b_blocknr==block)
175             return tmp;
176     return NULL;
177 }
178
179 /*
180  * Why like this, I hear you say... The reason is race-conditions.
181  * As we don't lock buffers (unless we are reading them, that is),
182  * something might happen to it while we sleep (ie a read-error
183  * will force it bad). This shouldn't really happen currently, but
184  * the code is ready.
185  */
186
187 // 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
188 // 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
189 // 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
190 // 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。
191
192 // 利用 hash 表在高速缓冲区中寻找指定的缓冲区块。若找到则对该缓冲区块上锁并返回块头指针。
193 struct buffer\_head * get\_hash\_table(int dev, int block)
194 {
195     struct buffer\_head * bh;
196
197     for (;;) {
198         // 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。
199         if (!(bh=find\_buffer(dev, block)))
200             return NULL;
201         // 对该缓冲区块增加引用计数，并等待该缓冲区块解锁（如果已被上锁）。由于经过了睡眠状态，
202         // 因此有必要再验证该缓冲区块的正确性，并返回缓冲区块头指针。
203         bh->b_count++;
204         wait\_on\_buffer(bh);
205         if (bh->b_dev == dev && bh->b_blocknr == block)
206             return bh;
207         // 如果在睡眠时该缓冲区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。
208         bh->b_count--;
209     }
210 }
211
212 /*
213  * Ok, this is getblk, and it isn't very clear, again to hinder
214  * race-conditions. Most of the code is seldom used, (ie repeating),
215  * so it should be much more efficient than it looks.
216  */

```

```

203 * The algorithm is changed: hopefully better, and an elusive bug removed.
204 */
205 /*
206  * 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
207  * 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
208  * 比看上去的样子有效得多。
209  *
210  * 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。
211  */
212 // 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
213 // 大。
214 #define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
215 //// 取高速缓冲中指定的缓冲块。
216 // 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,
217 // 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
218 // 新项。返回相应缓冲区头指针。
219 struct buffer_head * getblk(int dev, int block)
220 {
221     struct buffer_head * tmp, * bh;
222     repeat:
223     // 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。
224     if (bh = get_hash_table(dev, block))
225         return bh;
226     // 扫描空闲数据块链表, 寻找空闲缓冲区。
227     // 首先让 tmp 指向空闲链表的第一个空闲缓冲区头。
228     tmp = free_list;
229     do {
230     // 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,
231     // 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)。
232     // 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,
233     // 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
234     // 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,
235     // 因此此时 b_lock=1, 但 b_count=0。
236     if (tmp->b_count)
237         continue;
238     // 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
239     // 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
240     // 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
241     // 执行本循环, 看看能否找到一个 BADNESS() 最小的缓冲块。
242     if (!bh || BADNESS(tmp) < BADNESS(bh)) {
243         bh = tmp;
244         if (!BADNESS(tmp))
245             break;
246     }
247 /* and repeat until we find something good */ /* 重复操作直到找到适合的缓冲块 */
248 } while ((tmp = tmp->b_next_free) != free_list);
249 // 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
250 // 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
251 // 函数开始处重新查找空闲缓冲块。
252 if (!bh) {
253     sleep_on(&buffer_wait);
254     goto repeat; // 跳转至 210 行。

```



```

227     }
    // 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。
228     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。
267 struct buffer_head * bread(int dev, int block)
268 {
269     struct buffer_head * bh;
270
// 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。
271     if (!(bh=getblk(dev, block)))
272         panic("bread: getblk returned NULL\n");
273     if (bh->b_uptodate)
274         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。
275     ll_rw_block(READ, bh);
276     wait_on_buffer(bh);
277     if (bh->b_uptodate)
278         return bh;
279     brelse(bh);
280     return NULL;
281 }
282
///// 复制内存块。
// 从 from 地址复制一块（1024 字节）数据到 to 位置。
283 #define COPYBLK(from, to) \
284 __asm__ ("cld\n\t" \
285         "rep\n\t" \
286         "movsl\n\t" \
287         :: "c" (BLOCK_SIZE/4), "S" (from), "D" (to) \
288         : "cx", "di", "si")
289
290 /*
291  * bread_page reads four buffers into memory at the desired address. It's
292  * a function of its own, as there is some speed to be got by reading them
293  * all at the same time, not waiting for one to be read, and then another
294  * etc.
295  */
/*
 * 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）。
296 void bread_page(unsigned long address, int dev, int b[4])
297 {
298     struct buffer_head * bh[4];
299     int i;
300
301     // 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
302     // 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
303     // 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
304     // 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
305     // 随意读取 1—4 个数据块。
306     for (i=0 ; i<4 ; i++)
307     {
308         if (b[i]) { // 若块号有效。
309             if (bh[i] = getblk(dev, b[i]))
310                 if (!bh[i]->b_uptodate)
311                     ll_rw_block(READ, bh[i]);
312             } else
313                 bh[i] = NULL;
314
315     // 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
316     // 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
317     // 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。
318     for (i=0 ; i<4 ; i++, address += BLOCK_SIZE)
319     {
320         if (bh[i]) {
321             wait_on_buffer(bh[i]); // 等待缓冲块解锁（若被上锁的话）。
322             if (bh[i]->b_uptodate) // 若缓冲块中数据有效的则复制。
323                 COPYBLK((unsigned long) bh[i]->b_data, address);
324             brelse(bh[i]); // 释放该缓冲区。
325         }
326     }
327 }
328
329 /*
330  * Ok, breada can be used as bread, but additionally to mark other
331  * blocks for reading as well. End the argument list with a negative
332  * number.
333  */
334 /*
335  * 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
336  * 需要使用一个负数来表明参数列表的结束。
337  */
338 //// 从指定设备读取指定的一些块。
339 // 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
340 // NULL。
341 struct buffer_head * breada(int dev, int first, ...)
342 {
343     va_list args;
344     struct buffer_head * bh, *tmp;
345
346     // 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
347     // 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。
348     va_start(args, first);
349     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。
    // 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是马上就使用，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内存高端位置应该是
    // 640KB。否则缓冲区内内存高端一定大于 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

```

12.4 bitmap.c 程序

从本程序起，我们开始探讨文件系统得第 2 个部分，即文件系统底层操作函数部分。这部分共包括 5 个文件，分别是 `super.c`、`bitmap.c`、`truncate.c`、`inode.c` 和 `namei.c` 程序。

`super.c` 程序主要包含对文件系统超级块进行访问和管理的函数；`bitmap.c` 程序用于处理文件系统的逻辑块位图和 i 节点位图；`truncate.c` 程序仅有一个把文件数据长度截为 0 的函数 `truncate()`；`inode.c` 程序主要涉及文件系统 i 节点信息的访问和管理；`namei.c` 程序则主要用于完成从一个给定文件路径名寻找并加载其对应 i 节点信息的功能。

按照一个文件系统中各功能部分的顺序，我们应该按照上面给出程序名的顺序来分别对它们进行描述，但是由于 `super.c` 程序中另外还包含几个有关文件系统加载/卸载的高层函数或系统调用，需要使用到其他几个程序中的函数，因此我们把它放在介绍过 `inode.c` 程序之后再加以说明。

12.4.1 功能描述

本程序的功能和作用即简单又清晰，主要用于对文件系统的 i 节点位图和逻辑块位图进行占用和释放操作处理。i 节点位图的操作函数是 `free_inode()` 和 `new_inode()`，操作逻辑块位图的函数是 `free_block()` 和 `new_block()`。

函数 `free_block()` 用于释放指定设备 `dev` 上数据区中的逻辑块 `block`。具体操作是复位指定逻辑块 `block` 对应逻辑块位图中的比特位。它首先取指定设备 `dev` 的超级块，并根据超级块给出的设备数据逻辑块的范围，判断逻辑块号 `block` 的有效性。然后在高速缓冲区中进行查找，看看指定的逻辑块此时是否正在高速缓冲区中。若是，则将对应的缓冲块释放掉。接着计算 `block` 从数据区开始算起的数据逻辑块号（从 1 开始计数），并对逻辑块(区段)位图进行操作，复位对应的比特位。最后根据逻辑块号设置缓冲区中包含相应逻辑块位图缓冲块的已修改标志。

函数 `new_block()` 用于向设备 `dev` 申请一个逻辑块，返回逻辑块号，并置位指定逻辑块 `block` 对应的逻辑块位图比特位。它首先取指定设备 `dev` 的超级块。然后对整个逻辑块位图进行搜索，寻找首个是 0 的比特位。若没有找到，则说明盘设备空间已用完，函数返回 0。否则将找到的第 1 个 0 值比特位置 1，表示占用对应的数据逻辑块。并将包含该比特位的逻辑位图所在缓冲块的已修改标志置位。接着计算出数据逻辑块的盘块号，并在高速缓冲区中申请相应的缓冲块，并把该缓冲块清零。然后设置该缓冲块的已更新和已修改标志。最后释放该缓冲块，以便其他程序使用，并返回盘块号（逻辑块号）。

函数 `free_inode()` 用于释放指定的 `i` 节点，并复位对应的 `i` 节点位图比特位；`new_inode()` 用于为设备 `dev` 建立一个新 `i` 节点，并返回该新 `i` 节点的指针。主要操作过程是在内存 `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。这两个函数的处理过程与上述两个函数类似，因此这里不再赘述。

12.4.2 代码注释

程序 12-3 linux/fs/bitmap.c

```

1  /*
2   *  linux/fs/bitmap.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /* 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 */
8  #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。
9                               // 这里主要使用了其中的 memset() 函数。
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。
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
   // 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。
13 #define clear_block(addr) \
14     __asm__ ("cld\n\t"          // 清方向位。
15             "rep\n\t"          // 重复执行存储数据 (0)。
16             "stosl" \
17             :: "a" (0), "c" (BLOCK_SIZE/4), "D" ((long) (addr)): "cx", "di")
18
   // 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”。整个宏是一个语句表达式 (即圆括号括住的组合语句)，其
   // 值是组合语句中最后一条表达式语句 res (第 23 行) 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。
19 #define set_bit(nr, addr) ({\
20     register int res __asm__ ("ax"); \
21     __asm__ __volatile__ ("btsl %2, %3\n\tsetb %al": \
22     "=a" (res): "" (0), "r" (nr), "m" (*(addr))); \
23     res;})
24

```



```

//          brelse(bh);          // b_count--后即退出，该块还有人用。
//          return 0;
//      }
//      bh->b_dirt=0;          // 否则复位已修改和已更新标志。
//      bh->b_uptodate=0;
//      if (bh->b_count)          // 若此时 b_count 为 1，则调用 brelse() 释放之。
//          brelse(bh);
//  }
56     bh = get\_hash\_table(dev, block);
57     if (bh) {
58         if (bh->b_count != 1) {
59             printk("trying to free block (%04x:%d), count=%d\n",
60                 dev, block, bh->b_count);
61             return;
62         }
63         bh->b_dirt=0;          // 复位脏（已修改）标志位。
64         bh->b_uptodate=0;      // 复位更新标志。
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;    // 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         panic("free_block: bit already cleared");
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 }
74
////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。
75 int new\_block(int dev)
76 {
77     struct buffer\_head * bh;
78     struct super\_block * sb;
79     int i, j;
80
// 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）。
81     if (!(sb = get\_super(dev)))
82         panic("trying to get new block from nonexistant device");
83     j = 8192;
84     for (i=0 ; i<8 ; i++)
85         if (bh=sb->s_zmap[i])

```



```

86         if ((j=find_first_zero(bh->b_data))<8192)
87             break;
88     if (i>=8 || !bh || j>=8192)
89         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。
90     if (set_bit(j, bh->b_data))
91         panic("new_block: bit already set");
92     bh->b_dirt = 1;
93     j += i*8192 + sb->s_firstdatazone-1;
94     if (j >= sb->s_nzones)
95         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。
96     if (!(bh=getblk(dev, j)))
97         panic("new_block: cannot get block");
98     if (bh->b_count != 1)
99         panic("new_block: count is != 1");
100    clear_block(bh->b_data);
101    bh->b_uptodate = 1;
102    bh->b_dirt = 1;
103    brelse(bh);
104    return j;
105 }
106
    //// 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。
107 void free_inode(struct m_inode * inode)
108 {
109     struct super_block * sb;
110     struct buffer_head * bh;
111
    // 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。
112     if (!inode)
113         return;
114     if (!inode->i_dev) {
115         memset(inode, 0, sizeof(*inode));
116         return;
117     }
    // 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。
118     if (inode->i_count>1) {
119         printk("trying to free inode with count=%d\n", inode->i_count);

```

```

120         panic("free_inode");
121     }
122     if (inode->i_nlinks)
123         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。
124     if (!(sb = get_super(inode->i_dev)))
125         panic("trying to free inode on nonexistent device");
126     if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
127         panic("trying to free inode 0 or nonexistant inode");
128     if (!(bh=sb->s_imap[inode->i_num>>13]))
129         panic("nonexistent imap in superbblock");
    // 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。
130     if (clear_bit(inode->i_num&8191, bh->b_data))
131         printk("free_inode: bit already cleared. %d\n", i);
132     bh->b_dirt = 1;
133     memset(inode, 0, sizeof(*inode));
134 }
135
    // 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。
136 struct m_inode * new_inode(int dev)
137 {
138     struct m_inode * inode;
139     struct super_block * sb;
140     struct buffer_head * bh;
141     int i, j;
142
    // 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)。
143     if (!(inode=get_empty_inode())) // fs/inode.c, 第 194 行。
144         return NULL;
145     if (!(sb = get_super(dev))) // fs/super.c, 第 56 行。
146         panic("new_inode with unknown device");
147     j = 8192;
148     for (i=0 ; i<8 ; i++)
149         if (bh=sb->s_imap[i])
150             if ((j=find_first_zero(bh->b_data))<8192)
151                 break;
152     if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
153         iput(inode);
154         return NULL;
155     }
    // 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)。

```

```

156     if (set_bit(j, bh->b_data))
157         panic("new_inode: bit already set");
158     bh->b_dirt = 1;
159     inode->i_count=1;           // 引用计数。
160     inode->i_nlinks=1;         // 文件目录项链接数。
161     inode->i_dev=dev;          // i 节点所在的设备号。
162     inode->i_uid=current->euid; // i 节点所属用户 id。
163     inode->i_gid=current->egid; // 组 id。
164     inode->i_dirt=1;           // 已修改标志置位。
165     inode->i_num = j + i*8192; // 对应设备中的 i 节点号。
166     inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME; // 设置时间。
167     return inode;             // 返回该 i 节点指针。
168 }
169

```

12.5 truncate.c 程序

12.5.1 功能描述

本程序用于释放指定 i 节点在设备上占用的所有逻辑块，包括直接块、一次间接块和二次间接块。从而将文件的节点对应的文件长度截为 0，并释放占用的设备空间。i 节点中直接块和间接块的示意图见图 12-23 所示。

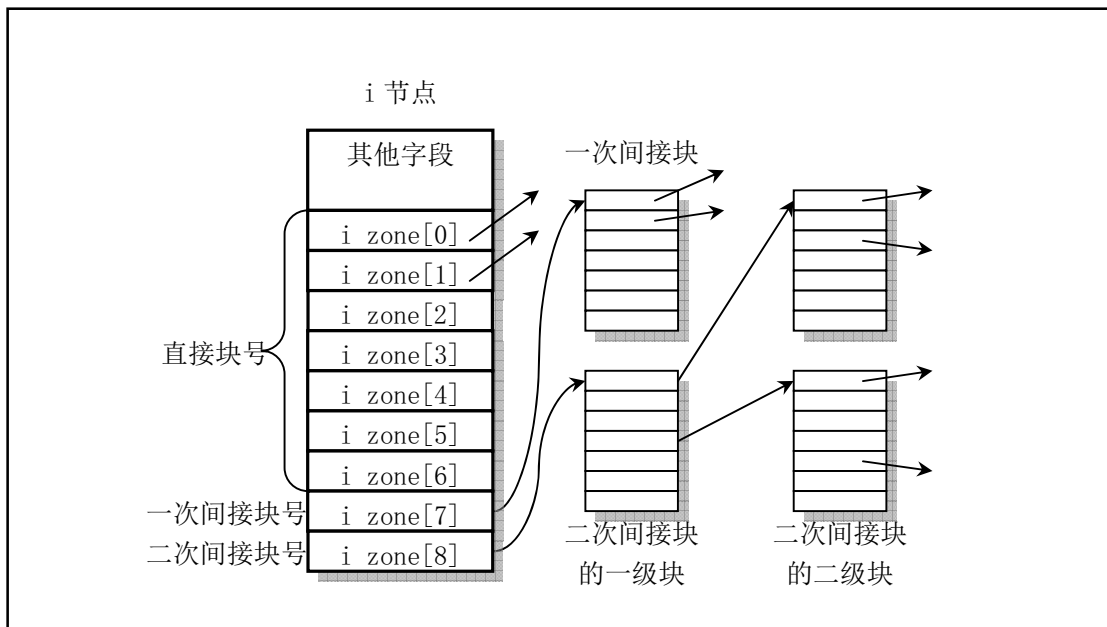


图 12-23 索引节点(i 节点)的逻辑块连接方式

12.5.2 代码注释

程序 12-4 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
11  // 释放所有一次间接块。（内部函数）
12  // 参数 dev 是文件系统所在设备的设备号；block 是逻辑块号。
13  static void free_ind(int dev, int block)
14  {
15      struct buffer_head * bh;
16      unsigned short * p;
17      int i;
18
19      // 首先判断参数的有效性。如果逻辑块号为 0，则返回。
20      if (!block)
21          return;
22      // 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。
23      // 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）。
24      if (bh=bread(dev, block)) {
25          p = (unsigned short *) bh->b_data; // 指向缓冲块数据区。
26          for (i=0; i<512; i++, p++) // 每个逻辑块上可有 512 个块号。
27              if (*p)
28                  free_block(dev, *p); // 释放指定的设备逻辑块。
29          brelse(bh); // 然后释放间接块占用的缓冲块。
30      }
31      // 最后释放设备上的一次间接块。
32      free_block(dev, block);
33  }
34
35  // 释放所有二次间接块。
36  // 参数 dev 是文件系统所在设备的设备号；block 是逻辑块号。
37  static void free_dind(int dev, int block)
38  {
39      struct buffer_head * bh;
40      unsigned short * p;
41      int i;
42
43      // 首先判断参数的有效性。如果逻辑块号为 0，则返回。
44      if (!block)
45          return;
46      // 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区。
47      if (bh=bread(dev, block)) {
48          p = (unsigned short *) bh->b_data; // 指向缓冲块数据区。
49          for (i=0; i<512; i++, p++) // 每个逻辑块上可连接 512 个二级块。
50              if (*p)
51                  free_ind(dev, *p); // 释放所有一次间接块。
52          brelse(bh); // 释放二次间接块占用的缓冲块。
53      }

```

```

// 最后释放设备上的二次间接块。
44     free_block(dev, block);
45 }
46
///// 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。
47 void truncate(struct m_inode * inode)
48 {
49     int i;
50
// 首先判断指定 i 节点有效性。如果不是常规文件或者是目录文件，则返回。
51     if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode)))
52         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）。
53     for (i=0; i<7; i++)
54         if (inode->i_zone[i]) { // 如果块号不为 0，则释放之。
55             free_block(inode->i_dev, inode->i_zone[i]);
56             inode->i_zone[i]=0;
57         }
58     free_ind(inode->i_dev, inode->i_zone[7]); // 释放所有一次间接块。
59     free_dind(inode->i_dev, inode->i_zone[8]); // 释放所有二次间接块。
60     inode->i_zone[7] = inode->i_zone[8] = 0; // 逻辑块项 7、8 置零。
61     inode->i_size = 0; // 文件大小置零。
62     inode->i_dirt = 1; // 置节点已修改标志。
// 最后重置文件修改时间和 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。
63     inode->i_mtime = inode->i_ctime = CURRENT_TIME;
64 }
65
66

```

12.6 inode.c 程序

12.6.1 功能描述

该程序主要包括处理 i 节点的函数 `iget()`、`iput()` 和块映射函数 `bmap()`，以及其他一些辅助函数。`iget()`、`iput()` 和 `bmap()` 主要用于 `namei.c` 程序中的由路径名寻找对应 i 节点的映射函数 `namei()`。

`iget()` 函数用于从设备 `dev` 上读取指定节点号 `nr` 的 i 节点，并且把节点的引用计数字段值 `i_count` 增 1。其操作流程见图 12-24 所示。该函数首先判断参数 `dev` 的有效性，并从 i 节点表中取一个空闲 i 节点。然后扫描 i 节点表，寻找指定节点号 `nr` 的 i 节点，并递增该 i 节点的引用次数。如果当前扫描的 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。否则说明已经找到指定设备号和节点号的 i 节点，就等待该节点解锁（如果已上锁的话）。在等待该节点解锁的阶段，节点表可能会发生变化，此时如果该 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，则需要再次重新扫描整个 i 节点表。随后把 i 节点的引用计数值增 1，并且判断该 i 节点是否是其他文件系统的安装点。

若该 i 节点是某个文件系统的安装点，则在超级块表中搜寻安装在此 i 节点的超级块。若没有找到

相应的超级块，则显示出错信息，并释放函数开始获取的空闲节点，返回该 i 节点指针。若找到了相应的超级块，则将该 i 节点写盘。再从安装在此 i 节点文件系统的超级块上取设备号，并令 i 节点号为 1。然后重新再次扫描整个 i 节点表，来取该被安装文件系统的根节点。

若该 i 节点不是其他文件系统的安装点，则说明已经找到了对应的 i 节点，因此此时可以放弃临时申请的空闲 i 节点，并返回找到的 i 节点指针。

如果在 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 节点在 i 节点表中建立该节点。并从相应设备上读取该 i 节点信息。返回该 i 节点指针。

`iput()`函数所完成的功能正好与 `iget()`相反，它主要用于把 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。如果 i 节点是块设备文件的 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，并在释放该 i 节点后返回。如果 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。则此时只要把 i 节点引用计数递减 1 后返回即可。因为 i 节点的 `i_count=0`，表示已释放。该函数所执行操作流程也与 `iget()`类似。

因此，若在某时刻进程不需要持续使用一个 i 节点时就应该调用 `iput()`函数来递减该 i 节点的引用计数字段 `i_count` 的值，同时也让内核执行其他一些处理。因此在执行过以下操作之一后，内核代码通常都应该调用 `iput()`函数：

- 把 i 节点引用计数字段 `i_count` 的值增 1；
- 调用了 `namei()`、`dir_namei()`或 `open_namei()`函数；
- 调用了 `iget()`、`new_inode()`或 `get_empty_inode()`函数；
- 在关闭一个文件时，若已经没有任何其他进程使用该文件；
- 卸载一个文件系统时（需要放回设备文件名 i 节点等）。

另外，一个进程被创建时，其当前工作目录 `pwd`、进程当前根目录 `root` 和可执行文件目录 `executalbe` 三个 i 节点结构指针字段都会被初始化而指向三个 i 节点，并且也相应地设置了这三个 i 节点的引用计数字段。因此，当进程执行改变当前工作目录的系统调用时，在该系统调用的代码中就需要调用 `iput()`函数来先放回原来使用中的 i 节点，然后再让进程的 `pwd` 指向新路径名的 i 节点。同样，若要修改进程的 `root` 和 `executable` 字段，那么也需要执行 `iput()`函数。

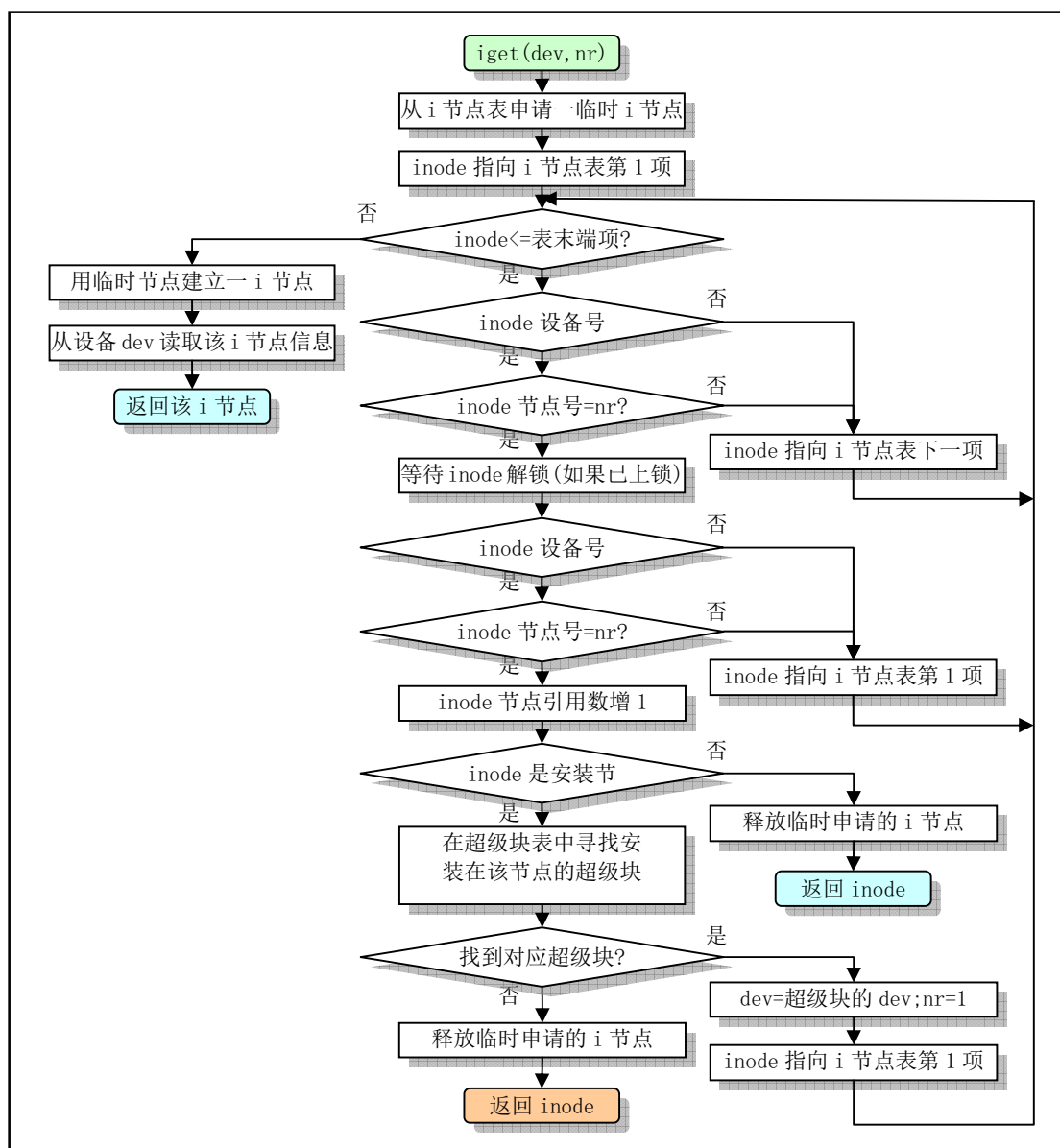


图 12-24 iget 函数操作流程

`_bmap()`函数用于把一个文件数据块映射到盘块的处理操作。所带的参数 `inode` 是文件的 `i` 节点指针，参数 `block` 是文件中的数据块号，参数 `create` 是创建标志，表示在对应文件数据块不存在的情况下，是否需要在盘上建立对应的盘块。该函数的返回值是文件数据块对应在设备上的逻辑块号（盘块号）。当 `create=0` 时，该函数就是 `bmap()`函数。当 `create=1` 时，它就是 `create_block()`函数。

正规文件中的数据是放在磁盘块的数据区中的，而一个文件名则通过对应的 `i` 节点与这些数据磁盘块相联系，这些盘块的号码就存放在 `i` 节点的逻辑块数组中。`_bmap()`函数主要是对 `i` 节点的逻辑块（区块）数组 `i_zone[]` 进行处理，并根据 `i_zone[]` 中所设置的逻辑块号（盘块号）来设置逻辑块位图的占用情况。参见“总体功能描述”一节中的图 12-6。正如前面所述，`i_zone[0]` 至 `i_zone[6]` 用于存放对应文件的直接逻辑块号；`i_zone[7]` 用于存放一次间接逻辑块号；而 `i_zone[8]` 用于存放二次间接逻辑块号。当文件较小时（小于 7K），就可以将文件所使用的盘块号直接存放在 `i` 节点的 7 个直接块项中；当文件稍大一些时（不超过 7K+512K），需要用到一次间接块项 `i_zone[7]`；当文件更大时，就需要用到二次间接块项 `i_zone[8]` 了。因此，文件比较小时，linux 寻址盘块的速度就比较快一些。

12.6.2 代码注释

程序 12-5 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
15 struct m_inode inode_table[NR_INODE]={0,},}; // 内存中 i 节点表 (NR_INODE=32 项)。
16
17 static void read_inode(struct m_inode * inode); // 读指定 i 节点号的 i 节点信息，294 行。
18 static void write_inode(struct m_inode * inode); // 写 i 节点信息到高速缓冲中，314 行。
19
    // 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。
20 static inline void wait_on_inode(struct m_inode * inode)
21 {
22     cli();
23     while (inode->i_lock)
24         sleep_on(&inode->i_wait); // kernel/sched.c, 第 151 行。
25     sti();
26 }
27
    // 对指定的 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。
28 static inline void lock_inode(struct m_inode * inode)
29 {
30     cli();
31     while (inode->i_lock)
32         sleep_on(&inode->i_wait);
33     inode->i_lock=1; // 置锁定标志。
34     sti();
35 }
36
    // 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。
37 static inline void unlock_inode(struct m_inode * inode)
38 {
39     inode->i_lock=0;
40     wake_up(&inode->i_wait); // kernel/sched.c, 第 188 行。

```

```

41 }
42
    // 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。
43 void invalidate_inodes(int dev)
44 {
45     int i;
46     struct m_inode * inode;
47
    // 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 48 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。
48     inode = 0+inode_table; // 指向 i 节点表指针数组首项。
49     for(i=0 ; i<NR_INODE ; i++,inode++) {
50         wait_on_inode(inode); // 等待该 i 节点可用（解锁）。
51         if (inode->i_dev == dev) {
52             if (inode->i_count) // 若其引用数不为 0，则显示出错警告。
53                 printk("inode in use on removed disk\n\r");
54             inode->i_dev = inode->i_dirt = 0; // 释放 i 节点(置设备号为 0)。
55         }
56     }
57 }
58
    // 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。
59 void sync_inodes(void)
60 {
61     int i;
62     struct m_inode * inode;
63
    // 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。
64     inode = 0+inode_table; // 让指针首先指向 i 节点表指针数组首项。
65     for(i=0 ; i<NR_INODE ; i++,inode++) { // 扫描 i 节点表指针数组。
66         wait_on_inode(inode); // 等待该 i 节点可用（解锁）。
67         if (inode->i_dirt && !inode->i_pipe) // 若 i 节点已修改且不是管道节点，
68             write_inode(inode); // 则写盘（实际是写入缓冲区中）。
69     }
70 }
71
    // 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 对应应在设备
    // 上的逻辑块号（盘块号）。
72 static int bmap(struct m_inode * inode,int block,int create)
73 {
74     struct buffer_head * bh;

```

```

75     int i;
76
77     // 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
78     // 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");
83     // 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
84     // 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
85     // 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。
86     // 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()
87     // 定义在 bitmap.c 程序中第 75 行开始处。
88     if (block < 7) {
89         if (create && !inode->i_zone[block])
90             if (inode->i_zone[block]=new_block(inode->i_dev)) {
91                 inode->i_ctime=CURRENT_TIME; // ctime - Change time.
92                 inode->i_dirt=1;              // 设置已修改标志。
93             }
94         return inode->i_zone[block];
95     }
96     // 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。
97     // 如果是创建，并且该 i 节点中对应该间接块字段 i_zone[7]是 0，表明文件是首次使用间接块，
98     // 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
99     // 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
100    // 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
101    // 接块，于是映射磁盘块失败，返回 0 退出。
102    block -= 7;
103    if (block < 512) {
104        if (create && !inode->i_zone[7])
105            if (inode->i_zone[7]=new_block(inode->i_dev)) {
106                inode->i_dirt=1;
107                inode->i_ctime=CURRENT_TIME;
108            }
109        if (!inode->i_zone[7])
110            return 0;
111        // 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
112        // 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，
113        // 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
114        // 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。
115        if (! (bh = bread(inode->i_dev, inode->i_zone[7])))
116            return 0;
117        i = ((unsigned short *) (bh->b_data))[block];
118        if (create && !i)
119            if (i=new_block(inode->i_dev)) {
120                ((unsigned short *) (bh->b_data))[block]=i;
121                bh->b_dirt=1;
122            }
123        // 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。
124        brelse(bh);
125        return i;
126    }

```

```

// 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。
109     block -= 512;
110     if (create && !inode->i_zone[8])
111         if (inode->i_zone[8]=new_block(inode->i_dev)) {
112             inode->i_dirt=1;
113             inode->i_ctime=CURRENT_TIME;
114         }
115     if (!inode->i_zone[8])
116         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。
117     if (! (bh=bread(inode->i_dev, inode->i_zone[8])))
118         return 0;
119     i = ((unsigned short *)bh->b_data)[block>>9];
120     if (create && !i)
121         if (i=new_block(inode->i_dev)) {
122             ((unsigned short *) (bh->b_data))[block>>9]=i;
123             bh->b_dirt=1;
124         }
125     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）。
126     if (!i)
127         return 0;
128     if (! (bh=bread(inode->i_dev, i)))
129         return 0;
130     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。
131     if (create && !i)
132         if (i=new_block(inode->i_dev)) {
133             ((unsigned short *) (bh->b_data))[block&511]=i;
134             bh->b_dirt=1;
135         }
// 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。
136     brelse(bh);
137     return i;
138 }
139
//// 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。
140 int bmap(struct m_inode * inode, int block)
141 {
142     return _bmap(inode, block, 0);
143 }
144
///// 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。返回设备上对应的已存在或新建的逻辑块号。
// 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。
145 int create_block(struct m_inode * inode, int block)
146 {
147     return _bmap(inode, block, 1);
148 }
149
///// 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。
150 void iput(struct m_inode * inode)
151 {
// 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。
152     if (!inode)
153         return;
154     wait_on_inode(inode);
155     if (!inode->i_count)
156         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()，228，234 行。
157     if (inode->i_pipe) {
158         wake_up(&inode->i_wait);
159         if (--inode->i_count)
160             return;
161         free_page(inode->i_size);
162         inode->i_count=0;
163         inode->i_dirt=0;
164         inode->i_pipe=0;
165         return;
166     }
// 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。
167     if (!inode->i_dev) {
168         inode->i_count--;
169         return;
170     }
// 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。
171     if (S_ISBLK(inode->i_mode)) {
172         sync_dev(inode->i_zone[0]);
173         wait_on_inode(inode);

```

```

174     }
    // 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 155 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。
175 repeat:
176     if (inode->i_count>1) {
177         inode->i_count--;
178         return;
179     }
180     if (!inode->i_nlinks) {
181         truncate(inode);
182         free_inode(inode);          // bitmap.c 第 107 行开始处。
183         return;
184     }
    // 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）。
185     if (inode->i_dirt) {
186         write_inode(inode);          /* we can sleep - so do again */
187         wait_on_inode(inode);        /* 因为我们睡眠了，所以需要重复判断 */
188         goto repeat;
189     }
    // 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。
190     inode->i_count--;
191     return;
192 }
193
194 // 从 i 节点表 (inode_table) 中获取一个空闲 i 节点项。
195 // 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。
196 struct m_inode * get_empty_inode(void)
197 {
198     struct m_inode * inode;
199     static struct m_inode * last_inode = inode_table;    // 指向 i 节点表第 1 项。
200     int i;
201     do {
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
203         // 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
204         // 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。
205         // 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
206         // i 节点，于是退出 for 循环。
207         inode = NULL;
208         for (i = NR_INODE; i ; i--) {          // NR_INODE = 32。
209             if (++last_inode >= inode_table + NR_INODE)
210                 last_inode = inode_table;
211             if (!last_inode->i_count) {
212                 inode = last_inode;
213                 if (!inode->i_dirt && !inode->i_lock)
214                     break;

```

```

209         }
210     }
    // 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。
    // [??]
211     if (!inode) {
212         for (i=0 ; i<NR_INODE ; i++)
213             printk("%04x: %6d\t", inode_table[i].i_dev,
214                 inode_table[i].i_num);
215             panic("No free inodes in mem");
216     }
    // 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。
217     wait_on_inode(inode);
218     while (inode->i_dirt) {
219         write_inode(inode);
220         wait_on_inode(inode);
221     }
    // 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。
222     } while (inode->i_count);
223     memset(inode, 0, sizeof(*inode));
224     inode->i_count = 1;
225     return inode;
226 }
227
    ///// 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。
228 struct m_inode * get_pipe_inode(void)
229 {
230     struct m_inode * inode;
231
    // 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。
232     if (!(inode = get_empty_inode()))
233         return NULL;
234     if (!(inode->i_size=get_free_page())) { // 节点的 i_size 字段指向缓冲区。
235         inode->i_count = 0;
236         return NULL;
237     }
    // 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。
238     inode->i_count = 2; /* sum of readers/writers */ /* 读/写两者总计 */
239     PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0; // 复位管道头尾指针。
240     inode->i_pipe = 1; // 置节点为管道使用的标志。
241     return inode;
242 }
243
    ///// 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 节点到内存 i 节点表中, 并返回该 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。
244 struct m_inode * iget(int dev, int nr)
245 {
246     struct m_inode * inode, * empty;
247
// 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。
248     if (!dev)
249         panic("iget with dev==0");
250     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。
251     inode = inode_table;
252     while (inode < NR_INODE+inode_table) {
253         if (inode->i_dev != dev || inode->i_num != nr) {
254             inode++;
255             continue;
256         }
// 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。
257         wait_on_inode(inode);
258         if (inode->i_dev != dev || inode->i_num != nr) {
259             inode = inode_table;
260             continue;
261         }
// 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。
262         inode->i_count++;
263         if (inode->i_mount) {
264             int i;
265
266             for (i = 0 ; i<NR_SUPER ; i++)
267                 if (super_block[i].s_imount==inode)
268                     break;
269             if (i >= NR_SUPER) {
270                 printk("Mounted inode hasn't got sb\n");
271                 if (empty)
272                     iput(empty);
273                 return inode;
274             }
// 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。
275         iput(inode);
276         dev = super_block[i].s_dev;
277         nr = ROOT_INO;

```

```

278             inode = inode\_table;
279             continue;
280         }
// 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。
281         if (empty)
282             iput(empty);
283         return inode;
284     }
// 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。
285     if (!empty)
286         return (NULL);
287     inode=empty;
288     inode->i_dev = dev;           // 设置 i 节点的设备。
289     inode->i_num = nr;           // 设置 i 节点号。
290     read\_inode(inode);
291     return inode;
292 }
293
//// 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。
294 static void read\_inode(struct m\_inode * inode)
295 {
296     struct super\_block * sb;
297     struct buffer\_head * bh;
298     int block;
299
// 首先锁定该 i 节点，并取该节点所在设备的超级块。
300     lock\_inode(inode);
301     if (!(sb=get\_super(inode->i_dev)))
302         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 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。
303     block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
304             (inode->i_num-1)/INODES\_PER\_BLOCK;
305     if (!(bh=bread(inode->i_dev, block)))
306         panic("unable to read i-node block");
307     *(struct d\_inode *)inode =
308         ((struct d\_inode *)bh->b_data)
309         [(inode->i_num-1)%INODES\_PER\_BLOCK];
// 最后释放读入的缓冲块，并解锁该 i 节点。
310     brelse(bh);
311     unlock\_inode(inode);

```

```

312 }
313
314 // 将 i 节点信息写入缓冲区中。
315 // 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
316 // 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
317 // 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
318 // 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。
319 static void write_inode(struct m_inode * inode)
320 {
321     struct super_block * sb;
322     struct buffer_head * bh;
323     int block;
324
325     // 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
326     // 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
327     // 获取该 i 节点的超级块。
328     lock_inode(inode);
329     if (!inode->i_dirt || !inode->i_dev) {
330         unlock_inode(inode);
331         return;
332     }
333     if (!(sb=get_super(inode->i_dev)))
334         panic("trying to write inode without device");
335     // 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
336     // 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
337     // 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。
338     block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
339         (inode->i_num-1)/INODES_PER_BLOCK;
340     if (!(bh=bread(inode->i_dev, block)))
341         panic("unable to read i-node block");
342     ((struct d_inode *)bh->b_data)
343         [(inode->i_num-1)%INODES_PER_BLOCK] =
344         *(struct d_inode *)inode;
345     // 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
346     // 释放该含有 i 节点的缓冲区，并解锁该 i 节点。
347     bh->b_dirt=1;
348     inode->i_dirt=0;
349     brelse(bh);
350     unlock_inode(inode);
351 }
352
353

```

12.6.3 其他信息

无☺。

12.7 super.c 程序

12.7.1 功能描述

该文件描述了对文件系统中超级块操作的函数，这些函数属于文件系统低层函数，供上层的文件名

和目录操作函数使用。主要有 `get_super()`、`put_super()` 和 `read_super()`。另外还有 2 个有关文件系统加载/卸载系统调用 `sys_umount()` 和 `sys_mount()`，以及根文件系统加载函数 `mount_root()`。其他一些辅助函数与 `buffer.c` 中的辅助函数的作用类似。

超级块中主要存放了有关整个文件系统的信息，其信息结构参见“总体功能描述”中的图 12-3。

`get_super()` 函数用于在指定设备的条件下，在内存超级块数组中搜索对应的超级块，并返回相应超级块的指针。因此，在调用该函数时，该相应的文件系统必须已经被加载（mount），或者起码该超级块已经占用了超级块数组中的一项，否则返回 NULL。

`put_super()` 用于释放指定设备的超级块。它把该超级块对应的文件系统的 i 节点位图和逻辑块位图所占用的缓冲块都释放掉，并释放超级块表（数组）`super_block[]` 中对应的操作块项。在调用 `umount()` 卸载一个文件系统或者更换磁盘时将会调用该函数。

`read_super()` 用于把指定设备的文件系统的超级块读入到缓冲区中，并登记到超级块表中，同时也把文件系统的 i 节点位图和逻辑块位图读入内存超级块结构的相应数组中。最后并返回该超级块结构的指针。

`sys_umount()` 系统调用用于卸载一个指定设备文件名的文件系统，而 `sys_mount()` 则用于往一个目录名上加载一个文件系统。

程序中最后一个函数 `mount_root()` 是用于安装系统的根文件系统，并将在系统初始化时被调用。其具体操作流程图 12-25 所示。

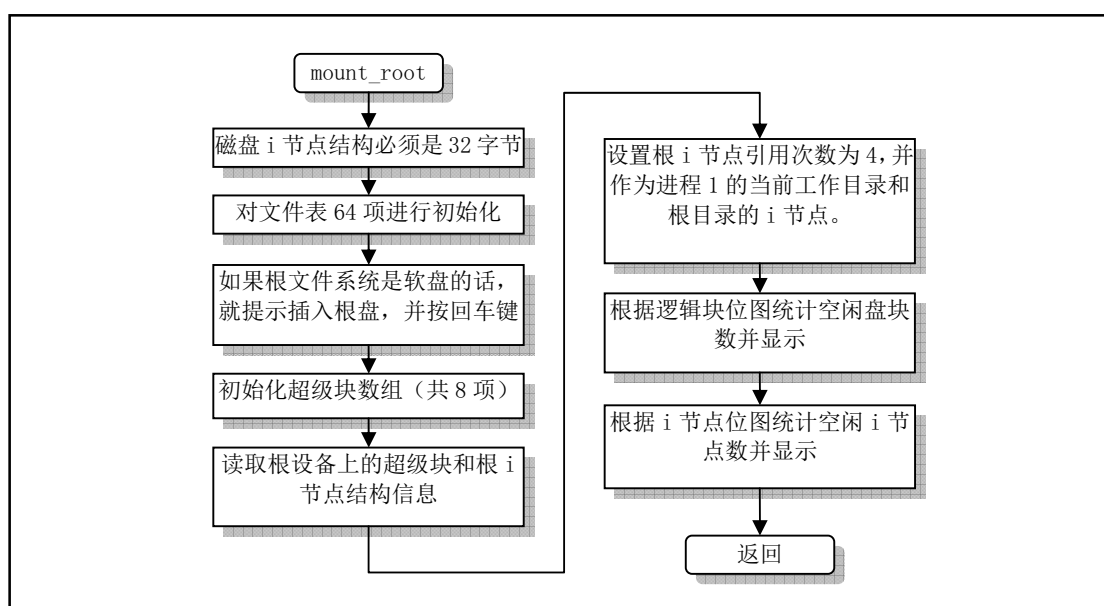


图 12-25 `mount_root()` 函数的功能

该函数除了用于安装系统的根文件系统以外，还对内核使用文件系统起到初始化的作用。它对内存中超级块数组进行了初始化，还对文件描述符数组表 `file_table[]` 进行了初始化，并对根文件系统中的空闲盘块数和空闲 i 节点数进行了统计并显示出来。

`mount_root()` 函数是在系统执行初始化程序 `main.c` 中，在进程 0 创建了第一个子进程（进程 1）后被调用的，而且系统仅在这里调用它一次。具体的调用位置是在初始化函数 `init()` 的 `setup()` 函数中。`setup()` 函数位于 `/kernel/blk_drv/hd.c` 第 71 行开始。

12.7.2 代码注释

程序 12-6 `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  */
10 /*
11  *  super.c 程序中含有处理超级块表的代码。
12  */
13 #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。
14 #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，
15 // 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/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/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
19 #include <errno.h> // 错误号头文件。包含系统中各种出错号。
20 #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。
21
22 // 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)。
23 int sync_dev(int dev);
24 // 等待击键 (kernel/chr_drv/tty_io.c, 140 行)。
25 void wait_for_keypress(void);
26
27 /* set_bit uses setb, as gas doesn't recognize setc */
28 /* set_bit() 使用了 setb 指令，因为汇编编译器 gas 不能识别指令 setc */
29 // 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)。
30 // 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。
31 // %0 - ax(__res), %1 - 0, %2 - bitnr, %3 - addr
32 // 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
33 // 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
34 // 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
35 // 位标志 CF 设置操作数 %al。如果 CF = 1 则 %al = 1，否则 %al = 0。
36 #define set_bit(bitnr, addr) ({ \
37     register int __res __asm__("ax"); \
38     __asm__("bt %2,%3;setb %%al": "=a" (__res): "a" (0), "r" (bitnr), "m" (*(addr))); \
39     __res; })
40
41 struct super_block super_block[NR_SUPER]; // 超级块结构表数组 (NR_SUPER = 8)。
42 /* this is initialized in init/main.c */
43 /* ROOT_DEV 已在 init/main.c 中被初始化 */
44 int ROOT_DEV = 0; // 根文件系统设备号。
45
46 // 以下 3 个函数 (lock_super()、free_super() 和 wait_on_super()) 的作用与 inode.c 文
47 // 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。
48 // 锁定超级块。
49 // 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
50 // 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。
51 static void lock_super(struct super_block * sb)
52 {
53     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 151 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 unlock\_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
74 // 释放（放回）指定设备的超级块。
75 // 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
76 // 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
77 // 装有其他文件系统，则不能释放该超级块。
78 void put_super(int dev)
79 {
80     struct super_block * sb;
81     struct m_inode * inode;
82     int i;
83
84     // 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
85     // 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
86     // 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
87     // 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，
88     // s_imount 会先被置成 Null 以后才会调用本函数，参见第 192 行。
89     if (dev == ROOT_DEV) {
90         printk("root diskette changed: prepare for armageddon\n\r");
91         return;
92     }
93     if (!(sb = get_super(dev)))
94         return;
95     if (sb->s_imount) {
96         printk("Mounted disk changed - tssk, tssk\n\r");
97         return;
98     }
99     // 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
100    // 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，
101    // 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
102    // 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
103    // 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
104    // 写入设备中。函数最后对该超级块解锁，并返回。
105    lock_super(sb);
106    sb->s_dev = 0; // 置超级块空闲。
107    for(i=0; i<I_MAP_SLOTS; i++)
108        brelse(sb->s_imap[i]);
109    for(i=0; i<Z_MAP_SLOTS; i++)
110        brelse(sb->s_zmap[i]);
111    free_super(sb);
112    return;
113 }
114
115 // 读取指定设备的超级块。
116 // 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。
117 // 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。
118 static struct super_block * read_super(int dev)
119 {
120     struct super_block * s;
121     struct buffer_head * bh;

```



```

104         int i, block;
105
106         // 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
107         // 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
108         // 均失效，需要进行失效处理，即释放原来加载的文件系统。
109         if (!dev)
110             return NULL;
111         check_disk_change(dev);
112         // 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
113         // 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。
114         if (s = get_super(dev))
115             return s;
116         for (s = 0+super_block ;; s++) {
117             if (s >= NR_SUPER+super_block)
118                 return NULL;
119             if (!s->s_dev)
120                 break;
121         }
122         // 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
123         // 该超级块结构中的内存字段进行部分初始化处理。
124         s->s_dev = dev;           // 用于 dev 设备上的文件系统。
125         s->s_isup = NULL;
126         s->s_imount = NULL;
127         s->s_time = 0;
128         s->s_rd_only = 0;
129         s->s_dirt = 0;
130         // 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
131         // 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
132         // 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
133         // 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
134         // 息的高速缓冲块。
135         lock_super(s);
136         if (!(bh = bread(dev, 1))) {           // 读 1 号磁盘块。磁盘块号从 0 号计起。
137             s->s_dev=0;
138             free_super(s);                     // unlock_super。
139             return NULL;
140         }
141         *((struct d_super_block *) s) =
142             *((struct d_super_block *) bh->b_data);
143         brelse(bh);
144         // 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
145         // 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，
146         // 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
147         // 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
148         // 数是 0x137f。
149         if (s->s_magic != SUPER_MAGIC) {
150             s->s_dev = 0;
151             free_super(s);
152             return NULL;
153         }
154         // 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。
155         // 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
156         // 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。
137     for (i=0; i<I_MAP_SLOTS; i++)                // 初始化操作。
138         s->s_imap[i] = NULL;
139     for (i=0; i<Z_MAP_SLOTS; i++)
140         s->s_zmap[i] = NULL;
141     block=2;
142     for (i=0 ; i < s->s_imap_blocks ; i++)          // 读取设备中 i 节点位图。
143         if (s->s_imap[i]=bread(dev, block))
144             block++;
145         else
146             break;
147     for (i=0 ; i < s->s_zmap_blocks ; i++)          // 读取设备中逻辑块位图。
148         if (s->s_zmap[i]=bread(dev, block))
149             block++;
150         else
151             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。
152     if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
153         for(i=0; i<I_MAP_SLOTS; i++)              // 释放位图占用的高速缓冲块。
154             brelse(s->s_imap[i]);
155         for(i=0; i<Z_MAP_SLOTS; i++)
156             brelse(s->s_zmap[i]);
157         s->s_dev=0;                                // 释放选定的超级块数组项。
158         free_super(s);                             // 解锁该超级块项。
159         return NULL;
160     }
// 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。
161     s->s_imap[0]->b_data[0] |= 1;
162     s->s_zmap[0]->b_data[0] |= 1;
163     free_super(s);
164     return s;
165 }
166
//// 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。
167 int sys_umount(char * dev_name)
168 {
169     struct m_inode * inode;
170     struct super_block * sb;
171     int dev;
172
// 首先根据设备文件名找到对应的 i 节点, 并取其其中的设备号。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。

```

```

173     if (!(inode=namei(dev_name)))
174         return -ENOENT;
175     dev = inode->i_zone[0];
176     if (!S_ISBLK(inode->i_mode)) {
177         iput(inode); // fs/inode.c, 第 150 行。
178         return -ENOTBLK;
179     }
    // 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。如果设备上根文件系统,
    // 则不能被卸载, 返回忙出错号。
180     iput(inode);
181     if (dev==ROOT_DEV)
182         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。
183     if (!(sb=get_super(dev)) || !(sb->s_imount))
184         return -ENOENT;
185     if (!sb->s_imount->i_mount)
186         printk("Mounted inode has i_mount=0\n");
187     for (inode=inode_table+0; inode<inode_table+NR_INODE; inode++)
188         if (inode->i_dev==dev && inode->i_count)
189             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。
190     sb->s_imount->i_mount=0;
191     iput(sb->s_imount);
192     sb->s_imount = NULL;
193     iput(sb->s_isup);
194     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)。
195     put_super(dev);
196     sync_dev(dev);
197     return 0;
198 }
199
    //// 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。
200 int sys_mount(char * dev_name, char * dir_name, int rw_flag)
201 {
202     struct m_inode * dev_i, * dir_i;
203     struct super_block * sb;
204     int dev;
205
    // 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。
206     if (!(dev_i=namei(dev_name)))

```

```

207         return -ENOENT;
208     dev = dev_i->i_zone[0];
209     if (!S_ISBLK(dev_i->i_mode)) {
210         iput(dev_i);
211         return -EPERM;
212     }
    // 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。
213     iput(dev_i);
214     if (!(dir_i=namei(dir_name)))
215         return -ENOENT;
216     if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
217         iput(dir_i);
218         return -EBUSY;
219     }
220     if (!S_ISDIR(dir_i->i_mode)) {                // 安装点需要是一个目录名。
221         iput(dir_i);
222         return -EPERM;
223     }
    // 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。
224     if (!(sb=read_super(dev))) {
225         iput(dir_i);
226         return -EBUSY;
227     }
    // 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。
228     if (sb->s_imount) {
229         iput(dir_i);
230         return -EBUSY;
231     }
232     if (dir_i->i_mount) {
233         iput(dir_i);
234         return -EPERM;
235     }
    // 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)。
236     sb->s_imount=dir_i;
237     dir_i->i_mount=1;
238     dir_i->i_dirt=1; /* NOTE! we don't iput(dir_i) */ /*注意!这里没用 iput(dir_i)*/
239     return 0;      /* we do that in umount */ /* 这将在 umount 内操作 */
240 }
241
    //// 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()) 被调用 (blk_drv/hd.c, 157 行)。
242 void mount_root(void)
243 {
244     int i, free;
245     struct super_block * p;
246     struct m_inode * mi;
247
// 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。
248     if (32 != sizeof (struct d_inode))
249         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。
250     for(i=0; i<NR_FILE; i++) // 初始化文件表。
251         file_table[i].f_count=0;
252     if (MAJOR(ROOT_DEV) == 2) { // 提示插入根文件系统盘。
253         printk("Insert root floppy and press ENTER");
254         wait_for_keypress();
255     }
256     for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
257         p->s_dev = 0; // 初始化超级块表。
258         p->s_lock = 0;
259         p->s_wait = NULL;
260     }
// 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。
261     if (!(p=read_super(ROOT_DEV)))
262         panic("Unable to mount root");
263     if (!(mi=iget(ROOT_DEV, ROOT_INO)) // 在 fs.h 中 ROOT_INO 定义为 1。
264         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)。
265     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 */
266     p->s_isup = p->s_imount = mi;
267     current->pwd = mi;
268     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。
269     free=0;
270     i=p->s_nzones;
271     while (-- i >= 0)
272         if (!set_bit(i&8191, p->s_zmap[i>>13]->b_data))
273             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。
274     printk("%d/%d free blocks\n", free, p->s_nzones);
275     free=0;
276     i=p->s_ninodes+1;
277     while (-- i >= 0)
278         if (!set_bit(i&8191, p->s_imap[i>>13]->b_data))
279             free++;
280     printk("%d/%d free inodes\n", free, p->s_ninodes);
281 }
282

```

12.8 namei.c 程序

12.8.1 功能描述

该文件是 Linux 0.11 内核中最长的函数，不过也只有 700 多行☺。本文件主要实现了根据目录名或文件名寻找到对应 i 节点的函数 `namei()`，以及一些关于目录的建立和删除、目录项的建立和删除等操作函数和系统调用。

Linux 0.11 系统采用的是 MINIX 文件系统 1.0 版。它的目录项结构与传统 UNIX 文件的目录项结构相同，定义在 `include/linux/fs.h` 文件中。在文件系统的一个目录中，其中所有文件名信息对应的目录项存储在该目录文件名的数据块中。例如，目录名 `root/` 下的所有文件名的目录项就保存在 `root/` 目录名文件的数据块中。而文件系统根目录下的所有文件名信息则保存在指定 i 节点（即 1 号节点）的数据块中。每个目录项只包括一个长度为 14 字节的文件名字符串和该文件名对应的 2 字节的 i 节点号。有关文件的其它信息则被保存在该 i 节点号指定的 i 节点结构中，该结构中主要包括文件访问属性、宿主、长度、访问保存时间以及所在磁盘块等信息。每个 i 节点号的 i 节点都位于磁盘上的固定位置处。

```

// 定义在 include/linux/fs.h 文件中。
36 #define NAME_LEN 14 // 名字长度值。
37 #define ROOT_INO 1 // 根 i 节点。

// 文件目录项结构。
157 struct dir_entry {
158     unsigned short inode; // i 节点号。
159     char name[NAME_LEN]; // 文件名。
160 };

```

在打开一个文件时，文件系统会根据给定的文件名找到其 i 节点号，从而找到文件所在的磁盘块位置。例如对于要查找文件名 `/usr/bin/vi` 的 i 节点号，文件系统首先会从具有固定 i 节点号（1）的根目录开始操作，即从 i 节点号 1 的数据块中查找到名称为 `usr` 的目录项，从而得到文件 `/usr` 的 i 节点号。根据该 i 节点号文件系统可以顺利地取得目录 `/usr`，并在其中可以查找到文件名 `bin` 的目录项。这样也就知道了 `/usr/bin` 的 i 节点号，因而我们可以知道目录 `/usr/bin` 的目录所在位置，并在该目录中查找到 `vi` 文件的目录项。最终我们获得了文件路径名 `/usr/bin/vi` 的 i 节点号，从而可以从磁盘上得到该 i 节点号的 i 节点结构信息。

在每个目录中还包含两个特殊的文件目录项，它们的名称分别固定是 `'.'` 和 `'..'`。 `'.'` 目录项中给出了当前

目录的 *i* 节点号，而 `..` 目录项中给出了当前目录的父目录的 *i* 节点号。因此在给出一个相对路径名时文件系统就可以利用这两个特殊目录项进行查找操作。例如要查找 `./kernel/Makefile`，就可以首先根据当前目录的 `..` 目录项得到父目录的 *i* 节点号，然后按照上面描述过程进行查找操作。

由于程序中几个主要函数的前面都有较详细的英文注释，而且各函数和系统调用的功能明了，所以这里就不再赘述。

12.8.2 代码注释

程序 12-7 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  *  tytso 作了一些纠正。
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 的数据，
14 // 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。
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 #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。
18 #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。
19 #include <errno.h> // 错误号头文件。包含系统中各种出错号。
20 #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。
21 #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。
22
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
24 // 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
25 // 的形式值 *(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
26 // 为 "LoveYou"[2]（或者 2["LoveYou"]）就等同于 *( "LoveYou" + 2)。另外，字符串 "LoveYou"
27 // 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
28 // 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
29 // 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
30 // 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）
31 // 就可用其 ASCII 码值来表示。
32 //
33 // 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。
34 // 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
35 // 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
36 // 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，
37 // 表示可读可写（rw）。另外，其中 0_ACCMODE = 00003，是索引值 x 的屏蔽码。
38 #define ACC_MODE(x) ("004\002\006\377"[ (x) & 0_ACCMODE ])
39
40 /*

```



```

24 * comment out this line if you want names > NAME_LEN chars to be
25 * truncated. Else they will be disallowed.
26 */
   /*
   * 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。
   */
27 /* #define NO_TRUNCATE */
28
29 #define MAY_EXEC 1           // 可执行(可进入)。
30 #define MAY_WRITE 2         // 可写。
31 #define MAY_READ 4          // 可读。
32
33 /*
34 *      permission()
35 *
36 * is used to check for read/write/execute permissions on a file.
37 * I don't know if we should look at just the euid or both euid and
38 * uid, but that should be easily changed.
39 */
   /*
   *      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。
40 static int permission(struct m_inode * inode, int mask)
41 {
42     int mode = inode->i_mode;           // 文件访问属性。
43
44 /* 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。
45     if (inode->i_dev && !inode->i_nlinks)
46         return 0;
47     else if (current->euid==inode->i_uid)
48         mode >>= 6;
49     else if (current->egid==inode->i_gid)
50         mode >>= 3;
   // 最后判断如果所取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。
51     if (((mode & mask & 0007) == mask) || suser())
52         return 1;
53     return 0;
54 }
55
56 /*
57 * ok, we cannot use strncmp, as the name is not in our data space.
58 * Thus we'll have to use match. No big problem. Match also makes
59 * some sanity tests.

```

```

60  *
61  * NOTE! unlike strcmp, match returns 1 for success, 0 for failure.
62  */
/*
 * ok, 我们不能使用 strcmp 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 strcmp 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 65 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。
63 static int match(int len, const char * name, struct dir_entry * de)
64 {
65     register int same __asm__("ax");
66
// 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。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, 也返回 0。
// 第 69 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。
67     if (!de || !de->inode || len > NAME_LEN)
68         return 0;
69     if (len < NAME_LEN && de->name[len])
70         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)。
71     __asm__("cld\n\t" // 清方向位。
72             "fs ; repe ; cmpsb\n\t" // 用户空间执行循环比较[esi++]和[edi++],
73             "setz %al" // 若结果一样(zf=0)则设置 al=1(same=eax)。
74             : "=a" (same)
75             : "" (0), "S" ((long) name), "D" ((long) de->name), "c" (len)
76             : "cx", "di", "si");
77     return same; // 返回比较结果。
78 }
79
80 /*
81  * find_entry()
82  *
83  * finds an entry in the specified directory with the wanted name. It
84  * returns the cache buffer in which the entry was found, and the entry
85  * itself (as a parameter - res_dir). It does NOT read the inode of the
86  * entry - you'll have to do that yourself if you want to.
87  *
88  * This also takes care of the few special cases due to '..'-traversal
89  * over a pseudo-root and a mount point.
90  */
/*

```

```

*   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。
91 static struct buffer head * find\_entry(struct m\_inode ** dir,
92     const char * name, int namelen, struct dir\_entry ** res_dir)
93 {
94     int entries;
95     int block,i;
96     struct buffer head * bh;
97     struct dir\_entry * de;
98     struct super\_block * sb;
99
// 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 27 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。
100 #ifdef NO_TRUNCATE
101     if (namelen > NAME\_LEN)
102         return NULL;
103 #else
104     if (namelen > NAME\_LEN)
105         namelen = NAME\_LEN;
106 #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。如果文件名长度等于 0，则返回 NULL，退出。
107     entries = (*dir)->i_size / (sizeof (struct dir\_entry));
108     *res_dir = NULL;
109     if (!namelen)
110         return 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:)
111 /* 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 */
112     if (namelen==2 && get\_fs\_byte(name)=='.' && get\_fs\_byte(name+1)=='.') {

```

```

113 /* '..' 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) */
114     if ((*dir) == current->root)
115         namelen=1;
116     else if ((*dir)->i_num == ROOT\_INO) {
117 /* '..' over a mount-point results in 'dir' being exchanged for the mounted
118    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 */
119         sb=get\_super((*dir)->i_dev);
120         if (sb->s_imount) {
121             iput(*dir);
122             (*dir)=sb->s_imount;
123             (*dir)->i_count++;
124         }
125     }
126 }

// 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。
127     if (!(block = (*dir)->i_zone[0]))
128         return NULL;
129     if (!(bh = bread((*dir)->i_dev, block)))
130         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。
131     i = 0;
132     de = (struct dir\_entry *) bh->b_data;
133     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 137 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。
134         if ((char *)de >= BLOCK\_SIZE+bh->b_data) {
135             brelse(bh);
136             bh = NULL;
137             if (!(block = bmap(*dir, i/DIR\_ENTRIES\_PER\_BLOCK)) ||
138                 !(bh = bread((*dir)->i_dev, block))) {
139                 i += DIR\_ENTRIES\_PER\_BLOCK;
140                 continue;
141             }
142             de = (struct dir\_entry *) bh->b_data;
143         }

// 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。
144         if (match(namelen, name, de)) {
145             *res_dir = de;
146             return bh;
147         }

```

```

148         de++;
149         i++;
150     }
    // 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）。
151     brelse(bh);
152     return NULL;
153 }
154
155 /*
156  *      add_entry()
157  *
158  * adds a file entry to the specified directory, using the same
159  * semantics as find_entry(). It returns NULL if it failed.
160  *
161  * NOTE!! The inode part of 'de' is left at 0 - which means you
162  * may not sleep between calling this and putting something into
163  * the entry, as someone else might have used it while you slept.
164  */
    /*
    *      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；
165 static struct buffer_head * add_entry(struct m_inode * dir,
166     const char * name, int namelen, struct dir_entry ** res_dir)
167 {
168     int block, i;
169     struct buffer_head * bh;
170     struct dir_entry * de;
171
    // 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 27 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。
172     *res_dir = NULL; // 用于返回目录项结构指针。
173 #ifdef NO_TRUNCATE
174     if (namelen > NAME_LEN)
175         return NULL;
176 #else
177     if (namelen > NAME_LEN)
178         namelen = NAME_LEN;
179 #endif
    // 现在我们将开始操作，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。
180     if (!namelen)
181         return NULL;
182     if (!(block = dir->i_zone[0]))
183         return NULL;
184     if (!(bh = bread(dir->i_dev, block)))
185         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。
186     i = 0;
187     de = (struct dir_entry *) bh->b_data;
188     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 192 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 145 行) 则可读取或创建出在设备上对应的逻辑块。
189         if ((char *)de >= BLOCK_SIZE+bh->b_data) {
190             brelse(bh);
191             bh = NULL;
192             block = create_block(dir, i/DIR_ENTRIES_PER_BLOCK);
193             if (!block)
194                 return NULL;
195             if (!(bh = bread(dir->i_dev, block))) { // 若空则跳过该块继续。
196                 i += DIR_ENTRIES_PER_BLOCK;
197                 continue;
198             }
199             de = (struct dir_entry *) bh->b_data;
200         }
// 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。
201         if (i*sizeof(struct dir_entry) >= dir->i_size) {
202             de->inode=0;
203             dir->i_size = (i+1)*sizeof(struct dir_entry);
204             dir->i_dirt = 1;
205             dir->i_ctime = CURRENT_TIME;
206         }
// 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。
207         if (!de->inode) {
208             dir->i_mtime = CURRENT_TIME;
209             for (i=0; i < NAME_LEN; i++)
210                 de->name[i]=(i<namelen)?get_fs_byte(name+i):0;

```



```

211             bh->b_dirt = 1;
212             *res_dir = de;
213             return bh;
214         }
215         de++;           // 如果该目录项已经被使用，则继续检测下一个目录项。
216         i++;
217     }
    // 本函数执行不到这里。这也许是 Linux 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☺。
218     brelse(bh);
219     return NULL;
220 }
221
222 /*
223  *      get_dir()
224  *
225  * Getdir traverses the pathname until it hits the topmost directory.
226  * It returns NULL on failure.
227  */
    /*
    *      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。
    */
    // 搜寻指定路径名的目录（或文件名）的 i 节点。
    // 参数：pathname - 路径名。
    // 返回：目录或文件的 i 节点指针。失败时返回 NULL。
228 static struct m_inode * get_dir(const char * pathname)
229 {
230     char c;
231     const char * thisname;
232     struct m_inode * inode;
233     struct buffer_head * bh;
234     int namelen, inr, idev;
235     struct dir_entry * de;
236
    // 搜索操作会从当前进程任务结构中设置的根（或伪根）i 节点或当前工作目录 i 节点开始。
    // 因此首先需要判断进程的根 i 节点指针和当前工作目录 i 节点指针是否有效。如果当前进程
    // 没有设定根 i 节点，或者该进程根 i 节点指向是一个空闲 i 节点（引用为 0），则系统出错
    // 停机。如果进程的当前工作目录 i 节点指针为空，或者该当前工作目录指向的 i 节点是一个
    // 空闲 i 节点，这也是系统有问题，停机。
237     if (!current->root || !current->root->i_count)
238         panic("No root inode");
239     if (!current->pwd || !current->pwd->i_count)
240         panic("No cwd inode");
    // 如果用户指定的路径名的第 1 个字符是 '/'，则说明路径名是绝对路径名。则从根 i 节点开
    // 始操作。否则若第一个字符是其他字符，则表示给定的是相对路径名。应从进程的当前工作
    // 目录开始操作。则取进程当前工作目录的 i 节点。如果路径名为空，则出错返回 NULL 退出。
    // 此时变量 inode 指向了正确的 i 节点 — 进程的根 i 节点或当前工作目录 i 节点之一。
241     if ((c=get_fs_byte(pathname))== '/') {
242         inode = current->root;
243         pathname++;

```



```

244     } else if (c)
245         inode = current->pwd;
246     else
247         return NULL;    /* empty name is bad */ /* 空的路径名是错误的 */
// 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。首先把得到的 i 节点引用计数
// 增 1，表示我们正在使用。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（或
// 文件名）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
// （或文件名）。如果该 i 节点不是目录类型的 i 节点，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，刚进入循环时，当前的 i 节点就是进程根 i 节
// 点或者是当前工作目录的 i 节点。
248     inode->i_count++;
249     while (1) {
250         thisname = pathname;
251         if (!S\_ISDIR(inode->i_mode) || !permission(inode, MAY\_EXEC)) {
252             iput(inode);
253             return NULL;
254         }
// 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。
255         for(namelen=0; (c=get\_fs\_byte(pathname++))&&(c!='/'); namelen++)
256             /* nothing */;
257         if (!c)
258             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）。
259         if (! (bh = find\_entry(&inode, thisname, namelen, &de))) {
260             iput(inode);
261             return NULL;
262         }
263         inr = de->inode;                                // 当前目录名部分的 i 节点号。
264         idev = inode->i_dev;
265         brelse(bh);
266         iput(inode);
267         if (!(inode = iget(idev, inr)))                  // 取 i 节点内容。
268             return NULL;
269     }
270 }
271
272 /*
273  *      dir_namei()
274  *
275  * dir_namei() returns the inode of the directory of the
276  * specified name, and the name within that directory.

```

```

277 */
    /*
    *    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。
    // 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。
278 static struct m_inode * dir_namei(const char * pathname,
279     int * namelen, const char ** name)
280 {
281     char c;
282     const char * basename;
283     struct m_inode * dir;
284
    // 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 255 行上的
    // “注意”说明。
285     if (!(dir = get_dir(pathname)))
286         return NULL;
287     basename = pathname;
288     while (c=get_fs_byte(pathname++))
289         if (c=='/')
290             basename=pathname;
291     *namelen = pathname-basename-1;
292     *name = basename;
293     return dir;
294 }
295
296 /*
297 *    namei()
298 *
299 * is used by most simple commands to get the inode of a specified name.
300 * Open, link etc use their own routines, but this is enough for things
301 * like 'chmod' etc.
302 */
    /*
    *    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。
    // 参数: pathname - 路径名。
    // 返回: 对应的 i 节点。
303 struct m_inode * namei(const char * pathname)
304 {
305     const char * basename;
306     int inr, dev, namelen;

```

```

307     struct m_inode * dir;
308     struct buffer_head * bh;
309     struct dir_entry * de;
310
311     // 首先查找指定路径的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。
312     // 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。因此我们
313     // 已经找到对应目录的 i 节点，可以直接返回该 i 节点退出。
314     if (!(dir = dir_namei(pathname, &namelen, &basename)))
315         return NULL;
316     if (!namelen)
317         /* special case: '/usr/' etc */
318         /* 对应于 '/usr/' 等情况 */
319         return dir;
320     // 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
321     // 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
322     // 返回 src/目录名的 i 节点。因为函数 dir_namei() 把不以 '/' 结束的最后一个名字当作一个
323     // 文件名来看待，所以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
324     // 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。
325     bh = find_entry(&dir, basename, namelen, &de);
326     if (!bh) {
327         iput(dir);
328         return NULL;
329     }
330     // 接着取该目录项的 i 节点号和设备号，并释放包含该目录项的高速缓冲块并放回目录 i 节点。
331     // 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
332     // 节点指针。
333     inr = de->inode;
334     dev = dir->i_dev;
335     brelse(bh);
336     iput(dir);
337     dir = iget(dev, inr);
338     if (dir) {
339         dir->i_atime = CURRENT_TIME;
340         dir->i_dirt = 1;
341     }
342     return dir;
343 }
344
345 /*
346  *      open_namei ()
347  *
348  *      namei for open - this is in fact almost the whole open-routine.
349  */
350
351 /*
352  *      open_namei ()
353  *
354  *      open() 函数使用的 namei 函数 - 这其实几乎是完整的打开文件程序。
355  */
356
357 // 文件打开 namei 函数。
358 // 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
359 // （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）、
360 // 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
361 // 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）、
362 // 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
363 // 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。参见相关文件 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。
337 int open_namei(const char * pathname, int flag, int mode,
338               struct m_inode ** res_inode)
339 {
340     const char * basename;
341     int inr, dev, namelen;
342     struct m_inode * dir, * inode;
343     struct buffer_head * bh;
344     struct dir_entry * de;
345
// 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 (O)，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 360 行上的注释。
346     if ((flag & O_TRUNC) && !(flag & O_ACCMODE))
347         flag |= O_WRONLY;
348     mode &= 0777 & ~current->umask;
349     mode |= I_REGULAR;
// 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。
350     if (!(dir = dir_namei(pathname, &namelen, &basename)))
351         return -ENOENT;
352     if (!namelen) {
353         /* special case: '/usr/' etc */
354         if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {
355             *res_inode=dir;
356             return 0;
357         }
358         iput(dir);
359         return -EISDIR;
360     }
// 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。
360     bh = find_entry(&dir, basename, namelen, &de);
361     if (!bh) {
362         if (!(flag & O_CREAT)) {
363             iput(dir);
364             return -ENOENT;
365         }
366         if (!permission(dir, MAY_WRITE)) {
367             iput(dir);
368             return -EACCES;
369         }
// 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。

```

```

370         inode = new_inode(dir->i_dev);
371         if (!inode) {
372             iput(dir);
373             return -ENOSPC;
374         }
375         inode->i_uid = current->euid;
376         inode->i_mode = mode;
377         inode->i_dirt = 1;
378         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。
379         if (!bh) {
380             inode->i_nlinks--;
381             iput(inode);
382             iput(dir);
383             return -ENOSPC;
384         }
385         de->inode = inode->i_num;
386         bh->b_dirt = 1;
387         brelse(bh);
388         iput(dir);
389         *res_inode = inode;
390         return 0;
391     }
// 若上面（360 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。
392     inr = de->inode;
393     dev = dir->i_dev;
394     brelse(bh);
395     iput(dir);
396     if (flag & O_EXCL)
397         return -EEXIST;
// 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。
398     if (!(inode = iget(dev, inr)))
399         return -EACCES;
400     if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
401         !permission(inode, ACC_MODE(flag))) {
402         iput(inode);
403         return -EPERM;
404     }
// 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）。
405     inode->i_atime = CURRENT_TIME;
406     if (flag & O_TRUNC)
407         truncate(inode);
408     *res_inode = inode;
409     return 0;

```

```

410 }
411
412 // 创建一个设备特殊文件或普通文件节点 (node)。
413 // 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
414 // 件或命名管道）。
415 // 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。
416 // 返回：成功则返回 0，否则返回出错码。
417 int sys_mknod(const char * filename, int mode, int dev)
418 {
419     const char * basename;
420     int namelen;
421     struct m_inode * dir, * inode;
422     struct buffer_head * bh;
423     struct dir_entry * de;
424
425     // 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
426     // 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
427     // 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
428     // 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
429     // 码退出。如果不是超级用户，则返回访问许可出错码。
430     if (!suser())
431         return -EPERM;
432     if (!(dir = dir_namei(filename, &namelen, &basename)))
433         return -ENOENT;
434     if (!namelen) {
435         iput(dir);
436         return -ENOENT;
437     }
438     if (!permission(dir, MAY_WRITE)) {
439         iput(dir);
440         return -EPERM;
441     }
442
443     // 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。
444     // 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
445     // 目录的 i 节点，返回文件已经存在的出错码退出。
446     bh = find_entry(&dir, basename, namelen, &de);
447     if (bh) {
448         brelse(bh);
449         iput(dir);
450         return -EEXIST;
451     }
452
453     // 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
454     // 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，
455     // 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
456     // 改时间、访问时间为当前时间，并设置 i 节点已修改标志。
457     inode = new_inode(dir->i_dev);
458     if (!inode) { // 若不成功则放回目录 i 节点，返回无空间出错码退出。
459         iput(dir);
460         return -ENOSPC;
461     }
462     inode->i_mode = mode;
463     if (S_ISBLK(mode) || S_ISCHR(mode))
464         inode->i_zone[0] = dev;

```