

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

10.1 总体功能	417
10.2 MAKEFILE 文件.....	427
10.3 KEYBOARD.S 程序	429
10.4 CONSOLE.C 程序.....	448
10.5 SERIAL.C 程序	474
10.6 RS_IO.S 程序	483
10.7 TTY_IO.C 程序.....	487
10.8 TTY_IOCTL.C 程序.....	499

第 11 章 数学协处理器(MATH)..... 507

11.1 MAKEFILE 文件.....	507
11.2 MATH-EMULATION.C 程序.....	509

第 12 章 文件系统(FS)..... 511

12.1 总体功能	511
12.2 MAKEFILE 文件.....	527
12.3 BUFFER.C 程序	530
12.4 BITMAP.C 程序.....	547
12.5 TRUNCT.C 程序.....	553
12.6 INODE.C 程序	555
12.7 SUPER.C 程序	567
12.8 NAMEI.C 程序	577
12.9 FILE_TABLE.C 程序.....	601
12.10 BLOCK_DEV.C 程序	601
12.11 FILE_DEV.C 程序.....	605
12.12 PIPE.C 程序.....	608
12.13 CHAR_DEV.C 程序	612
12.14 READ_WRITE.C 程序.....	615
12.15 OPEN.C 程序	621
12.16 EXEC.C 程序	627
12.17 STAT.C 程序	647
12.18 FCNTL.C 程序	649
12.19 IOCTL.C 程序.....	652

第 13 章 内存管理(MM)..... 655

13.1 总体功能	655
13.2 MAKEFILE 文件.....	661
13.3 MEMORY.C 程序.....	662
13.4 PAGE.S 程序.....	679

第 14 章 头文件(INCLUDE) 683

14.1 INCLUDE/目录下的文件	683
14.2 A.OUT.H 文件.....	684
14.3 CONST.H 文件	695
14.4 CTYPE.H 文件	695
14.5 ERRNO.H 文件	697
14.6 FCNTL.H 文件	699
14.7 SIGNAL.H 文件	701
14.8 STDARG.H 文件.....	703
14.9 STDDEF.H 文件	704
14.10 STRING.H 文件	705
14.11 TERMIOS.H 文件	715

14.12 TIME.H 文件.....	722
14.13 UNISTD.H 文件.....	724
14.14 UTIME.H 文件	729
14.15 INCLUDE/ASM/目录下的文件	731
14.16 IO.H 文件	731
14.17 MEMORY.H 文件.....	732
14.18 SEGMENT.H 文件.....	733
14.19 SYSTEM.H 文件.....	735
14.20 INCLUDE/LINUX/目录下的文件	739
14.21 CONFIG.H 文件	739
14.22 FDREG.H 头文件	741
14.23 FS.H 文件.....	744
14.24 HDREG.H 文件.....	749
14.25 HEAD.H 文件	752
14.26 KERNEL.H 文件.....	753
14.27 MM.H 文件.....	754
14.28 SCHED.H 文件.....	754
14.29 SYS.H 文件	761
14.30 TTY.H 文件.....	763
14.31 INCLUDE/SYS/目录中的文件.....	766
14.32 STAT.H 文件	766
14.33 TIMES.H 文件.....	767
14.34 TYPES.H 文件.....	768
14.35 UTSNAME.H 文件.....	769
14.36 WAIT.H 文件.....	770

第 15 章 库文件(LIB)..... 773

15.1 MAKEFILE 文件	774
15.2 _EXIT.C 程序	776
15.3 CLOSE.C 程序	777
15.4 CTYPE.C 程序	777
15.5 DUP.C 程序	778
15.6 ERRNO.C 程序.....	779
15.7 EXECVE.C 程序.....	779
15.8 MALLOC.C 程序	780
15.9 OPEN.C 程序	789
15.10 SETSID.C 程序.....	790
15.11 STRING.C 程序.....	791
15.12 WAIT.C 程序.....	791
15.13 WRITE.C 程序	792

第 16 章 建造工具(TOOLS)..... 795

16.1 BUILD.C 程序.....	795
----------------------	-----

第 17 章 实验环境设置与使用方法 802

17.1 BOCHS 仿真系统	802
17.2 在 BOCHS 中运行 LINUX 0.11 系统.....	806
17.3 访问磁盘映像文件中的信息.....	813
17.4 编译运行简单内核示例程序.....	815
17.5 利用 BOCHS 调试内核.....	817
17.6 创建磁盘映像文件.....	824
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		

```

446     inode->i_mtime = inode->i_atime = CURRENT\_TIME;
447     inode->i_dirt = 1;
// 接着为这个新的 i 节点在目录中新添加一个目录项。如果失败（包含该目录项的高速缓冲
// 块指针为 NULL），则放回目录的 i 节点；把所申请的 i 节点引用连接计数复位，并放回该
// i 节点，返回出错码退出。
448     bh = add\_entry(dir, basename, namelen, &de);
449     if (!bh) {
450         iput(dir);
451         inode->i_nlinks=0;
452         iput(inode);
453         return -ENOSPC;
454     }
// 现在添加目录项操作也成功了，于是我们来设置这个目录项内容。令该目录项的 i 节点字
// 段等于新 i 节点号，并置高速缓冲区已修改标志，放回目录和新的 i 节点，释放高速缓冲
// 区，最后返回 0(成功)。
455     de->inode = inode->i_num;
456     bh->b_dirt = 1;
457     iput(dir);
458     iput(inode);
459     brelse(bh);
460     return 0;
461 }
462
//// 创建一个目录。
// 参数：pathname - 路径名；mode - 目录使用的权限属性。
// 返回：成功则返回 0，否则返回出错码。
463 int sys\_mkdir(const char * pathname, int mode)
464 {
465     const char * basename;
466     int namelen;
467     struct m\_inode * dir, * inode;
468     struct buffer\_head * bh, *dir_block;
469     struct dir\_entry * de;
470
// 首先检查操作许可和参数的有效性并取路径名中顶层目录的 i 节点。如果不是超级用户，则
// 返回访问许可出错码。如果找不到对应路径名中顶层目录的 i 节点，则返回出错码。如果最
// 顶端的文件名长度为 0，则说明给出的路径名最后没有指定文件名，放回该目录 i 节点，返
// 回出错码退出。如果在该目录中没有写的权限，则放回该目录的 i 节点，返回访问许可出错
// 码退出。如果不是超级用户，则返回访问许可出错码。
471     if (!suser())
472         return -EPERM;
473     if (!(dir = dir\_namei(pathname, &namelen, &basename)))
474         return -ENOENT;
475     if (!namelen) {
476         iput(dir);
477         return -ENOENT;
478     }
479     if (!permission(dir, MAY\_WRITE)) {
480         iput(dir);
481         return -EPERM;
482     }
// 然后我们搜索一下路径名指定的目录名是否已经存在。若已经存在则不能创建同名目录节点。
// 如果对应路径名上最后的目录名的目录项已经存在，则释放包含该目录项的缓冲区块并放回

```

```

// 目录的 i 节点，返回文件已经存在的出错码退出。否则我们就申请一个新的 i 节点，并设置
// 该 i 节点的属性模式：置该新 i 节点对应的文件长度为 32 字节（2 个目录项的大小）、置
// 节点已修改标志，以及节点的修改时间和访问时间。2 个目录项分别用于 '.' 和 '..' 目录。
483     bh = find\_entry(&dir, basename, namelen, &de);
484     if (bh) {
485         brelse(bh);
486         iput(dir);
487         return -EEXIST;
488     }
489     inode = new\_inode(dir->i_dev);
490     if (!inode) { // 若不成功则放回目录的 i 节点，返回无空间出错码。
491         iput(dir);
492         return -ENOSPC;
493     }
494     inode->i_size = 32;
495     inode->i_dirt = 1;
496     inode->i_mtime = inode->i_atime = CURRENT\_TIME;
// 接着为该新 i 节点申请一用于保存目录项数据的磁盘块，用于保存目录项结构信息。并令 i
// 节点的第一个直接块指针等于该块号。如果申请失败则放回对应目录的 i 节点；复位新申请
// 的 i 节点连接计数；放回该新的 i 节点，返回没有空间出错码退出。否则置该新的 i 节点已
// 修改标志。
497     if (!(inode->i_zone[0]=new\_block(inode->i_dev))) {
498         iput(dir);
499         inode->i_nlinks--;
500         iput(inode);
501         return -ENOSPC;
502     }
503     inode->i_dirt = 1;
// 从设备上读取新申请的磁盘块（目的是把对应块放到高速缓冲区中）。若出错，则放回对应
// 目录的 i 节点；释放申请的磁盘块；复位新申请的 i 节点连接计数；放回该新的 i 节点，返
// 回没有空间出错码退出。
504     if (!(dir_block=bread(inode->i_dev, inode->i_zone[0]))) {
505         iput(dir);
506         free\_block(inode->i_dev, inode->i_zone[0]);
507         inode->i_nlinks--;
508         iput(inode);
509         return -ERROR;
510     }
// 然后我们在缓冲块中建立起所创建目录文件中的 2 个默认的新目录项（'.' 和 '..'）结构数
// 据。首先令 de 指向存放目录项的数据块，然后置该目录项的 i 节点号字段等于新申请的 i
// 节点号，名字字段等于 "."。然后 de 指向下一个目录项结构，并在该结构中存放上级目录
// 的 i 节点号和名字 ".."。然后设置该高速缓冲块已修改标志，并释放该缓冲块。再初始化
// 设置新 i 节点的模式字段，并置该 i 节点已修改标志。
511     de = (struct dir\_entry *) dir_block->b_data;
512     de->inode=inode->i_num; // 设置 '.' 目录项。
513     strcpy(de->name, ".");
514     de++;
515     de->inode = dir->i_num; // 设置 '..' 目录项。
516     strcpy(de->name, "..");
517     inode->i_nlinks = 2;
518     dir_block->b_dirt = 1;
519     brelse(dir_block);
520     inode->i_mode = I\_DIRECTORY | (mode & 0777 & ~current->umask);

```

```

521     inode->i_dirt = 1;
// 现在我们在指定目录中新添加一个目录项，用于存放新建目录的 i 节点号和目录名。如果
// 失败（包含该目录项的高速缓冲区指针为 NULL），则放回目录的 i 节点；所申请的 i 节点
// 引用连接计数复位，并放回该 i 节点。返回出错码退出。
522     bh = add\_entry(dir, basename, namelen, &de);
523     if (!bh) {
524         iput(dir);
525         free\_block(inode->i_dev, inode->i_zone[0]);
526         inode->i_nlinks=0;
527         iput(inode);
528         return -ENOSPC;
529     }
// 最后令该新目录项的 i 节点字段等于新 i 节点号，并置高速缓冲块已修改标志，放回目录
// 和新的 i 节点，释放高速缓冲块，最后返回 0（成功）。
530     de->inode = inode->i_num;
531     bh->b_dirt = 1;
532     dir->i_nlinks++;
533     dir->i_dirt = 1;
534     iput(dir);
535     iput(inode);
536     brelse(bh);
537     return 0;
538 }
539
540 /*
541  * routine to check that the specified directory is empty (for rmdir)
542  */
543 /*
544  * 用于检查指定的目录是否为空的子程序（用于 rmdir 系统调用）。
545  */
546 // 检查指定目录是否空。
547 // 参数：inode - 指定目录的 i 节点指针。
548 // 返回：1 - 目录中是空的；0 - 不空。
549 static int empty\_dir(struct m\_inode * inode)
550 {
551     int nr, block;
552     int len;
553     struct buffer\_head * bh;
554     struct dir\_entry * de;
555
556     // 首先计算指定目录中现有目录项个数并检查开始两个特定目录项中信息是否正确。一个目录
557     // 中应该起码有 2 个目录项：即“.”和“..”。如果目录项个数少于 2 个或者该目录 i 节点的第
558     // 1 个直接块没有指向任何磁盘块号，或者该直接块读不出，则显示警告信息“设备 dev 上目
559     // 录错”，返回 0(失败)。
560     len = inode->i_size / sizeof (struct dir\_entry); // 目录中目录项个数。
561     if (len<2 || !inode->i_zone[0] ||
562         !(bh=bread(inode->i_dev, inode->i_zone[0]))) {
563         printk("warning - bad directory on dev %04x\n", inode->i_dev);
564         return 0;
565     }
566     // 此时 bh 所指缓冲块中含有目录项数据。我们让目录项指针 de 指向缓冲块中第 1 个目录项。
567     // 对于第 1 个目录项（“.”），它的 i 节点号字段 inode 应该等于当前目录的 i 节点号。对于
568     // 第 2 个目录项（“..”），它的 i 节点号字段 inode 应该等于上一层目录的 i 节点号，不会

```

```

// 为 0。因此如果第 1 个目录项的 i 节点号字段值不等于该目录的 i 节点号，或者第 2 个目录
// 项的 i 节点号字段为零，或者两个目录项的名字字段不分别等于"."和"..", 则显示出错警
// 告信息“设备 dev 上目录错”，并返回 0。
556     de = (struct dir\_entry *) bh->b_data;
557     if (de[0].inode != inode->i_num || !de[1].inode ||
558         strcmp(".",de[0].name) || strcmp("..",de[1].name)) {
559         printk("warning - bad directory on dev %04x\n",inode->i_dev);
560         return 0;
561     }
// 然后我们令 nr 等于目录项序号（从 0 开始计）；de 指向第三个目录项。并循环检测该目录
// 中其余所有的（len - 2）个目录项，看有没有目录项的 i 节点号字段不为 0（被使用）。
562     nr = 2;
563     de += 2;
564     while (nr<len) {
// 如果该块磁盘块中的目录项已经全部检测完毕，则释放该磁盘块的缓冲块，并读取目录数据
// 文件中下一块含有目录项的磁盘块。读取的方法是根据当前检测的目录项序号 nr 计算出对
// 应目录项在目录数据文件中的数据块号（nr/DIR_ENTRIES_PER_BLOCK），然后使用 bmap()
// 函数取得对应的盘块号 block，再使用读设备盘块函数 bread() 把相应盘块读入缓冲块中，
// 并返回该缓冲块的指针。若所读取的相应盘块没有使用（或已经不用，如文件已经删除等），
// 则继续读下一块，若读不出，则出错返回 0。否则让 de 指向读出块的首个目录项。
565         if ((void *) de >= (void *) (bh->b_data+BLOCK\_SIZE)) {
566             brelse(bh);
567             block=bmap(inode,nr/DIR\_ENTRIES\_PER\_BLOCK);
568             if (!block) {
569                 nr += DIR\_ENTRIES\_PER\_BLOCK;
570                 continue;
571             }
572             if (!(bh=bread(inode->i_dev,block)))
573                 return 0;
574             de = (struct dir\_entry *) bh->b_data;
575         }
// 对于 de 指向的当前目录项，如果该目录项的 i 节点号字段不等于 0，则表示该目录项目前正
// 被使用，则释放该高速缓冲区，返回 0 退出。否则，若还没有查询完该目录中的所有目录项，
// 则把目录项序号 nr 增 1、de 指向下一个目录项，继续检测。
576         if (de->inode) {
577             brelse(bh);
578             return 0;
579         }
580         de++;
581         nr++;
582     }
// 执行到这里说明该目录中没有找到已用的目录项(当然除了头两个以外)，则释放缓冲块返回 1。
583     brelse(bh);
584     return 1;
585 }
586
///// 删除目录。
// 参数： name - 目录名（路径名）。
// 返回：返回 0 表示成功，否则返回出错号。
587 int sys\_rmdir(const char * name)
588 {
589     const char * basename;
590     int namelen;

```

```

591     struct m\_inode * dir, * inode;
592     struct buffer head * bh;
593     struct dir\_entry * de;
594
    // 首先检查操作许可和参数的有效性并取路径名中顶层目录的 i 节点。如果不是超级用户，则
    // 返回访问许可出错码。如果找不到对应路径名中顶层目录的 i 节点，则返回出错码。如果最
    // 顶端的文件名长度为 0，则说明给出的路径名最后没有指定目录名，放回该目录 i 节点，返
    // 回出错码退出。如果在该目录中没有写的权限，则放回该目录的 i 节点，返回访问许可出错
    // 码退出。如果不是超级用户，则返回访问许可出错码。
595     if (!suser())
596         return -EPERM;
597     if (!(dir = dir\_namei(name, &namelen, &basename)))
598         return -ENOENT;
599     if (!namelen) {
600         iput(dir);
601         return -ENOENT;
602     }
603     if (!permission(dir, MAY\_WRITE)) {
604         iput(dir);
605         return -EPERM;
606     }
    // 然后根据指定目录的 i 节点和目录名利用函数 find_entry() 寻找对应目录项，并返回包含该
    // 目录项的缓冲块指针 bh、包含该目录项的目录的 i 节点指针 dir 和该目录项指针 de。再根据
    // 该目录项 de 中的 i 节点号利用 iget() 函数得到对应的 i 节点 inode。如果对应路径名上最
    // 后目录名的目录项不存在，则释放包含该目录项的高速缓冲区，放回目录的 i 节点，返回文
    // 件已经存在出错码，并退出。如果取目录项的 i 节点出错，则放回目录的 i 节点，并释放含
    // 有目录项的高速缓冲区，返回出错号。
607     bh = find\_entry(&dir, basename, namelen, &de);
608     if (!bh) {
609         iput(dir);
610         return -ENOENT;
611     }
612     if (!(inode = iget(dir->i_dev, de->inode))) {
613         iput(dir);
614         brelse(bh);
615         return -EPERM;
616     }
    // 此时我们已有包含要被删除目录项的目录 i 节点 dir、要被删除目录项的 i 节点 inode 和要
    // 被删除目录项指针 de。下面我们通过对这 3 个对象中信息的检查来验证删除操作的可行性。

    // 若该目录设置了受限删除标志并且进程的有效用户 id (euid) 不是 root，并且进程的有效
    // 用户 id (euid) 不等于该 i 节点的用户 id，则表示当前进程没有权限删除该目录，于是放
    // 回包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，然后释放高速缓冲区，返回
    // 出错码。
617     if ((dir->i_mode & S\_ISVTX) && current->euid &&
618         inode->i_uid != current->euid) {
619         iput(dir);
620         iput(inode);
621         brelse(bh);
622         return -EPERM;
623     }
    // 如果要被删除的目录项 i 节点的设备号不等于包含该目录项的目录的设备号，或者该被删除
    // 目录的引用连接计数大于 1（表示有符号连接等），则不能删除该目录。于是释放包含要删

```



```

// 除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲块，返回出错码。
624     if (inode->i_dev != dir->i_dev || inode->i_count>1) {
625         iput(dir);
626         iput(inode);
627         brelse(bh);
628         return -EPERM;
629     }
// 如果要被删除目录的目录项 i 节点就等于包含该需删除目录的目录 i 节点，则表示试图删除
// “.”目录，这是不允许的。于是放回包含要删除目录名的目录 i 节点和要删除目录的 i 节点，
// 释放高速缓冲块，返回出错码。
630     if (inode == dir) {      /* we may not delete ".", but "../dir" is ok */
631         iput(inode);      /* 我们不可以删除“.”，但可以删除“../dir” */
632         iput(dir);
633         brelse(bh);
634         return -EPERM;
635     }
// 若要被删除目录 i 节点的属性表明这不是一个目录，则本删除操作的前提完全不存在。于是
// 放回包含删除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲块，返回出错码。
636     if (!ISDIR(inode->i_mode)) {
637         iput(inode);
638         iput(dir);
639         brelse(bh);
640         return -ENOTDIR;
641     }
// 若该需被删除的目录不空，则也不能删除。于是放回包含要删除目录名的目录 i 节点和该要
// 删除目录的 i 节点，释放高速缓冲块，返回出错码。
642     if (!empty\_dir(inode)) {
643         iput(inode);
644         iput(dir);
645         brelse(bh);
646         return -ENOTEMPTY;
647     }
// 对于一个空目录，其目录项链接数应该为 2（链接到上层目录和本目录）。若该需被删除目
// 录的 i 节点的连接数不等于 2，则显示警告信息。但删除操作仍然继续执行。于是置该需被
// 删除目录的目录项的 i 节点号字段为 0，表示该目录项不再使用，并置含有该目录项的高速
// 缓冲块已修改标志，并释放该缓冲块。然后再置被删除目录 i 节点的链接数为 0（表示空闲），
// 并置 i 节点已修改标志。
648     if (inode->i_nlinks != 2)
649         printk("empty directory has nlink!=2 (%d)", inode->i_nlinks);
650     de->inode = 0;
651     bh->b_dirt = 1;
652     brelse(bh);
653     inode->i_nlinks=0;
654     inode->i_dirt=1;
// 再将包含被删除目录名的目录的 i 节点链接计数减 1，修改其改变时间和修改时间为当前时
// 间，并置该节点已修改标志。最后放回包含要删除目录名的目录 i 节点和该要删除目录的 i
// 节点，返回 0（删除操作成功）。
655     dir->i_nlinks--;
656     dir->i_ctime = dir->i_mtime = CURRENT\_TIME;
657     dir->i_dirt=1;
658     iput(dir);
659     iput(inode);
660     return 0;

```

```

661 }
662
663 // 删除（释放）文件名对应的目录项。
664 // 从文件系统删除一个名字。如果是文件的最后一个链接，并且没有进程正打开该文件，则该
665 // 文件也将被删除，并释放所占用的设备空间。
666 // 参数：name - 文件名（路径名）。
667 // 返回：成功则返回 0，否则返回出错号。
668 int sys_unlink(const char * name)
669 {
670     const char * basename;
671     int namelen;
672     struct m_inode * dir, * inode;
673     struct buffer_head * bh;
674     struct dir_entry * de;
675
676     // 首先检查参数的有效性并取路径名中顶层目录的 i 节点。如果找不到对应路径名中顶层目录
677     // 的 i 节点，则返回出错码。如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指
678     // 定文件名，放回该目录 i 节点，返回出错码退出。如果在该目录中没有写的权限，则放回该
679     // 目录的 i 节点，返回访问许可出错码退出。如果找不到对应路径名顶层目录的 i 节点，则返
680     // 回出错码。
681     if (!(dir = dir_namei(name, &namelen, &basename)))
682         return -ENOENT;
683     if (!namelen) {
684         iput(dir);
685         return -ENOENT;
686     }
687     if (!permission(dir, MAY_WRITE)) {
688         iput(dir);
689         return -EPERM;
690     }
691
692     // 然后根据指定目录的 i 节点和目录名利用函数 find_entry() 寻找对应目录项，并返回包含该
693     // 目录项的缓冲块指针 bh、包含该目录项的目录的 i 节点指针 dir 和该目录项指针 de。再根据
694     // 该目录项 de 中的 i 节点号利用 iget() 函数得到对应的 i 节点 inode。如果对应路径名上最
695     // 后目录名的目录项不存在，则释放包含该目录项的高速缓冲区，放回目录的 i 节点，返回文
696     // 件已经存在出错码，并退出。如果取目录项的 i 节点出错，则放回目录的 i 节点，并释放含
697     // 有目录项的高速缓冲区，返回出错号。
698     bh = find_entry(&dir, basename, namelen, &de);
699     if (!bh) {
700         iput(dir);
701         return -ENOENT;
702     }
703     if (!(inode = iget(dir->i_dev, de->inode))) {
704         iput(dir);
705         brelse(bh);
706         return -ENOENT;
707     }
708
709     // 此时我们已有包含要被删除目录项的目录 i 节点 dir、要被删除目录项的 i 节点 inode 和要
710     // 被删除目录项指针 de。下面我们通过对这 3 个对象中信息的检查来验证删除操作的可行性。
711
712     // 若该目录设置了受限删除标志并且进程的有效用户 id (euid) 不是 root，并且进程的 euid
713     // 不等于该 i 节点的用户 id，并且进程的 euid 也不等于目录 i 节点的用户 id，则表示当前进
714     // 程没有权限删除该目录，于是放回包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，
715     // 然后释放高速缓冲块，返回出错码。

```

```

691     if ((dir->i_mode & S_ISVTX) && !suser() &&
692         current->euid != inode->i_uid &&
693         current->euid != dir->i_uid) {
694         iput(dir);
695         iput(inode);
696         brelse(bh);
697         return -EPERM;
698     }
    // 如果该指定文件名是一个目录，则也不能删除。放回该目录 i 节点和该文件名目录项的 i 节
    // 点，释放包含该目录项的缓冲块，返回出错号。
699     if (S_ISDIR(inode->i_mode)) {
700         iput(inode);
701         iput(dir);
702         brelse(bh);
703         return -EPERM;
704     }
    // 如果该 i 节点的链接计数值已经为 0，则显示警告信息，并修正其为 1。
705     if (!inode->i_nlinks) {
706         printk("Deleting nonexistent file (%04x:%d), %d\n",
707             inode->i_dev, inode->i_num, inode->i_nlinks);
708         inode->i_nlinks=1;
709     }
    // 现在我们可以删除文件名对应的目录项了。于是将该文件名目录项中的 i 节点号字段置为 0，
    // 表示释放该目录项，并设置包含该目录项的缓冲块已修改标志，释放该高速缓冲块。
710     de->inode = 0;
711     bh->b_dirt = 1;
712     brelse(bh);
    // 然后把文件名对应 i 节点的链接数减 1，置已修改标志，更新改变时间为当前时间。最后放
    // 回该 i 节点和目录的 i 节点，返回 0（成功）。如果是文件的最后一个链接，即 i 节点链接
    // 数减 1 后等于 0，并且此时没有进程正打开该文件，那么在调用 iput() 放回 i 节点时，该文
    // 件也将被删除，并释放所占用的设备空间。参见 fs/inode.c，第 180 行。
713     inode->i_nlinks--;
714     inode->i_dirt = 1;
715     inode->i_ctime = CURRENT_TIME;
716     iput(inode);
717     iput(dir);
718     return 0;
719 }
720
    ///// 为文件建立一个文件名目录项。
    // 为一个已存在的文件创建一个新链接（也称为硬连接 - hard link）。
    // 参数：oldname - 原路径名；newname - 新的路径名。
    // 返回：若成功则返回 0，否则返回出错号。
721 int sys_link(const char * oldname, const char * newname)
722 {
723     struct dir_entry * de;
724     struct m_inode * oldinode, * dir;
725     struct buffer_head * bh;
726     const char * basename;
727     int namelen;
728
    // 首先对原文件名进行有效性验证，它应该存在并且不是一个目录名。所以我们先取原文件路
    // 径名对应的 i 节点 oldinode。如果为 0，则表示出错，返回出错号。如果原路径名对应的是

```

```

// 一个目录名，则放回该 i 节点，也返回出错号。
729     oldinode=namei(oldname);
730     if (!oldinode)
731         return -ENOENT;
732     if (S_ISDIR(oldinode->i_mode)) {
733         iput(oldinode);
734         return -EPERM;
735     }
// 然后查找新路径名的最顶层目录的 i 节点 dir，并返回最后的文件名及其长度。如果目录的
// i 节点没有找到，则放回原路径名的 i 节点，返回出错号。如果新路径名中不包括文件名，
// 则放回原路径名 i 节点和新路径名目录的 i 节点，返回出错号。
736     dir = dir_namei(newname, &namelen, &basename);
737     if (!dir) {
738         iput(oldinode);
739         return -EACCES;
740     }
741     if (!namelen) {
742         iput(oldinode);
743         iput(dir);
744         return -EPERM;
745     }
// 我们不能跨设备建立硬链接。因此如果新路径名顶层目录的设备号与原路径名的设备号
// 不一样，则放回新路径名目录的 i 节点和原路径名的 i 节点，返回出错号。另外，如果用户
// 没有在新目录中写的权限，则也不能建立连接，于是放回新路径名目录的 i 节点和原路径名
// 的 i 节点，返回出错号。
746     if (dir->i_dev != oldinode->i_dev) {
747         iput(dir);
748         iput(oldinode);
749         return -EXDEV;
750     }
751     if (!permission(dir, MAY_WRITE)) {
752         iput(dir);
753         iput(oldinode);
754         return -EACCES;
755     }
// 现在查询该新路径名是否已经存在，如果存在则也不能建立链接。于是释放包含该已存在目
// 录项的高速缓冲块，放回新路径名目录的 i 节点和原路径名的 i 节点，返回出错号。
756     bh = find_entry(&dir, basename, namelen, &de);
757     if (bh) {
758         brelse(bh);
759         iput(dir);
760         iput(oldinode);
761         return -EEXIST;
762     }
// 现在所有条件都满足了，于是我们在新目录中添加一个目录项。若失败则放回该目录的 i 节
// 点和原路径名的 i 节点，返回出错号。否则初始设置该目录项的 i 节点号等于原路径名的 i
// 节点号，并置包含该新添目录项的缓冲块已修改标志，释放该缓冲块，放回目录的 i 节点。
763     bh = add_entry(dir, basename, namelen, &de);
764     if (!bh) {
765         iput(dir);
766         iput(oldinode);
767         return -ENOSPC;
768     }

```

```

769     de->inode = oldinode->i_num;
770     bh->b_dirt = 1;
771     brelse(bh);
772     iput(dir);
    // 再将原节点的链接计数加 1，修改其改变时间为当前时间，并设置 i 节点已修改标志。最后
    // 放回原路径名的 i 节点，并返回 0（成功）。
773     oldinode->i_nlinks++;
774     oldinode->i_ctime = CURRENT_TIME;
775     oldinode->i_dirt = 1;
776     iput(oldinode);
777     return 0;
778 }
779

```

12.9 file_table.c 程序

12.9.1 功能描述

该程序目前是空的，仅定义了文件表数组。

12.9.2 代码注释

程序 12-8 linux/fs/file_table.c

```

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

```

12.10 block_dev.c 程序

从这里开始是文件系统程序的第 3 部分。包括 5 个程序：block_dev.c、char_dev.c、pipe.c、file_dev.c 和 read_write.c。前 4 个程序为 read_write.c 提供服务，主要实现了文件系统的数据访问操作。read_write.c 程序主要实现了系统调用 sys_write()和 sys_read()。这 5 个程序可以看作是系统调用与块设备、字符设备、管道“设备”和文件系统“设备”的接口驱动程序。它们之间的关系可以用图 12-26 表示。系统调用 sys_write()或 sys_read()会根据参数所提供文件描述符的属性，判断出是哪种类型的文件，然后分别调用相应设备接口程序中的读/写函数，而这些函数随后会执行相应的驱动程序。

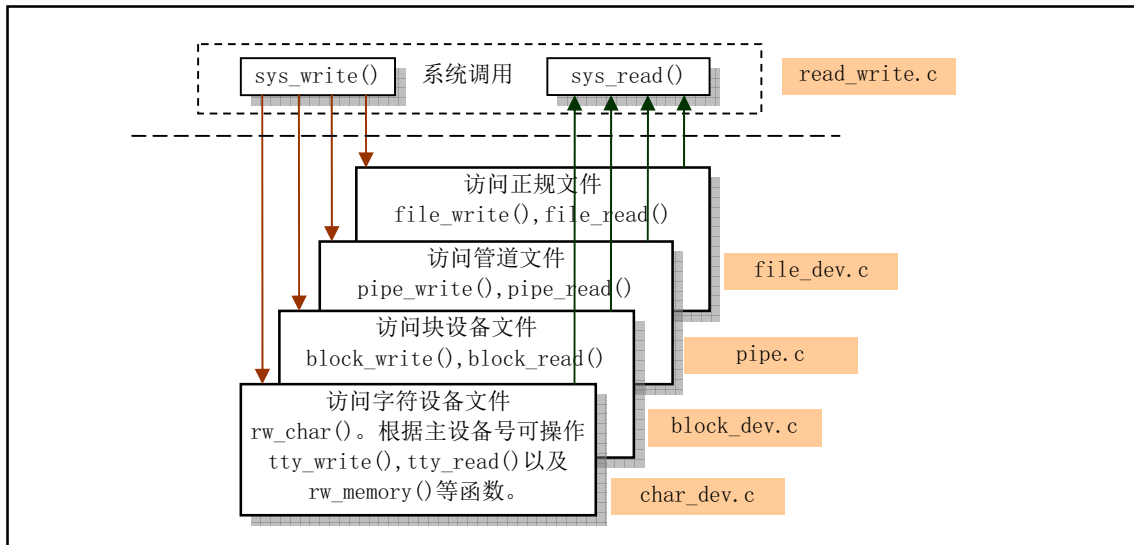


图 12-26 各种类型文件与文件系统和系统调用的接口函数

12.10.1 功能描述

`block_dev.c` 程序属于块设备文件数据访问操作类程序。该文件包括 `block_read()` 和 `block_write()` 两个块设备读写函数,分别用来直接读写块设备上的原始数据。这两个函数是供系统调用函数 `read()` 和 `write()` 调用,其他地方没有引用。

由于块设备每次对磁盘读写是以盘块为单位(与缓冲区中缓冲块长度相同),因此函数 `block_write()` 首先把参数中文件指针 `pos` 位置映射成数据块号和块中偏移量值,然后使用块读取函数 `bread()` 或块预读函数 `breada()` 将文件指针位置所在的数据块读入缓冲区的一个缓冲块中,然后根据本块中需要写的数据长度 `chars`,从用户数据缓冲中将数据复制到当前缓冲块的偏移位置开始处。如果还有需要写的数据,则再将下一块读入缓冲区的缓冲块中,并将用户数据复制到该缓冲块中,在第二次及以后写数据时,偏移量 `offset` 均为 0。参见图 12-27 所示。

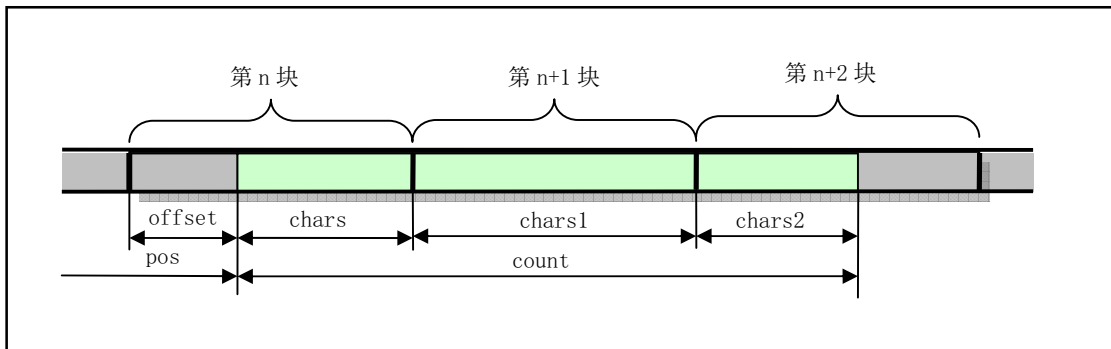


图 12-27 块数据读写操作指针位置示意图

用户的缓冲区是用户程序在开始执行时由系统分配的,或者是在执行过程中动态申请的。用户缓冲区使用的虚拟线性地址,在调用本函数之前,系统会将虚拟线性地址映射到主内存区中相应的内存页中。

函数 `block_read()` 的操作方式与 `block_write()` 相同,只是把数据从缓冲区复制到用户指定的地方。

12.10.2 代码注释

程序 12-9 linux/fs/block_dev.c

```

1  /*
2   *  linux/fs/block_dev.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <errno.h>          // 错误号头文件。包含系统中各种出错号。
8
9  #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                               // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
10 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
11 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
12 #include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
13
14  // 数据块写函数 - 向指定设备从给定偏移处写入指定长度数据。
15  // 参数: dev - 设备号; pos - 设备文件中偏移量指针; buf - 用户空间中缓冲区地址;
16  // count - 要传送的字节数。
17  // 返回已写入字节数。若没有写入任何字节或出错，则返回出错号。
18  // 对于内核来说，写操作是向高速缓冲区中写入数据。什么时候数据最终写入设备是由高速缓
19  // 冲管理程序决定并处理的。另外，因为块设备是以块为单位进行读写，因此对于写开始位置
20  // 不处于块起始处时，需要先将开始字节所在的整个块读出，然后将需要写的数据从写开始处
21  // 填写满该块，再将完整的一块数据写盘（即交由高速缓冲程序去处理）。
22
23  int block_write(int dev, long * pos, char * buf, int count)
24  {
25      // 首先由文件中位置 pos 换算成开始读写盘块的块序号 block，并求出需写第 1 字节在该块中
26      // 的偏移位置 offset。
27
28      int block = *pos >> BLOCK_SIZE_BITS;          // pos 所在文件数据块号。
29      int offset = *pos & (BLOCK_SIZE-1);           // pos 在数据块中偏移值。
30      int chars;
31      int written = 0;
32      struct buffer_head * bh;
33      register char * p;                             // 局部寄存器变量，被存放在寄存器中。
34
35      // 然后针对要写入的字节数 count，循环执行以下操作，直到数据全部写入。在循环执行过程
36      // 中，先计算在当前处理的数据块中可写入的字节数。如果需要写入的字节数填不满一块，那
37      // 么就只需写 count 字节。如果正好要写 1 块数据内容，则直接申请 1 块高速缓冲块，并把用
38      // 户数据放入即可。否则就需要读入将被写入部分数据的数据块，并预读下两块数据。然后将
39      // 块号递增 1，为下次操作做好准备。如果缓冲块操作失败，则返回已写字节数，如果没有写
40      // 入任何字节，则返回出错号（负数）。
41
42      while (count>0) {
43          chars = BLOCK_SIZE - offset;               // 本块可写入的字节数。
44          if (chars > count)
45              chars=count;
46          if (chars == BLOCK_SIZE)
47              bh = getblk(dev,block);                // buffer.c 第 206、322 行。
48          else
49              bh = breada(dev,block,block+1,block+2,-1);
50          block++;
51          if (!bh)

```



```

33         return written?written:-EIO;
// 接着先把指针 p 指向读出数据的缓冲块中开始写入数据的位置处。若最后一次循环写入的数
// 据不足一块，则需从块开始处填写（修改）所需的字节，因此这里需预先设置 offset 为零。
// 此后将文件中偏移指针 pos 前移此次将要写的字节数 chars，并累加这些要写的字节数到统
// 计值 written 中。再把还需要写的计数值 count 减去此次要写的字节数 chars。然后我们从
// 用户缓冲区复制 chars 个字节到 p 指向的高速缓冲块中开始写入的位置处。复制完后就设置
// 该缓冲区块已修改标志，并释放该缓冲区（也即该缓冲区引用计数递减 1）。
34         p = offset + bh->b_data;
35         offset = 0;
36         *pos += chars;
37         written += chars;           // 累计写入字节数。
38         count -= chars;
39         while (chars-->0)
40             *(p++) = get_fs_byte(buf++);
41         bh->b_dirt = 1;
42         brelse(bh);
43     }
44     return written;                // 返回已写入的字节数，正常退出。
45 }
46
//// 数据块读函数 - 从指定设备和位置处读入指定长度数据到用户缓冲区中。
// 参数: dev - 设备号; pos - 设备文件中偏移量指针; buf - 用户空间中缓冲区地址;
// count - 要传送的字节数。
// 返回已读入字节数。若没有读入任何字节或出错，则返回出错号。
47 int block_read(int dev, unsigned long * pos, char * buf, int count)
48 {
// 首先由文件中位置 pos 换算成开始读写盘块的块序号 block，并求出需读第 1 字节在该块中
// 的偏移位置 offset。
49     int block = *pos >> BLOCK_SIZE_BITS;
50     int offset = *pos & (BLOCK_SIZE-1);
51     int chars;
52     int read = 0;
53     struct buffer_head * bh;
54     register char * p;             // 局部寄存器变量，被存放在寄存器中。
55
// 然后针对要读入的字节数 count，循环执行以下操作，直到数据全部读入。在循环执行过程
// 中，先计算在当前处理的数据块中需读入的字节数。如果需要读入的字节数还不满一块，那
// 么就只需读 count 字节。然后调用读块函数 breada() 读入需要的数据块，并预读下两块数据，
// 如果读操作出错，则返回已读字节数，如果没有读入任何字节，则返回出错号。然后将块号
// 递增 1。为下次操作做好准备。如果缓冲块操作失败，则返回已写字节数，如果没有读入任
// 何字节，则返回出错号（负数）。
56     while (count>0) {
57         chars = BLOCK_SIZE-offset;
58         if (chars > count)
59             chars = count;
60         if (!(bh = breada(dev, block, block+1, block+2, -1)))
61             return read?read:-EIO;
62         block++;
// 接着先把指针 p 指向读出盘块的缓冲块中开始读入数据的位置处。若最后一次循环读操作的
// 数据不足一块，则需从块起始处读取所需字节，因此这里需预先设置 offset 为零。此后将
// 文件中偏移指针 pos 前移此次将要读的字节数 chars，并且累加这些要读的字节数到统计值
// read 中。再把还需要读的计数值 count 减去此次要读的字节数 chars。然后我们从高速缓冲
// 块中 p 指向的开始读的位置处复制 chars 个字节到用户缓冲区中，同时把用户缓冲区指针前

```

```

// 移。本次复制完后就释放该缓冲块。
63         p = offset + bh->b_data;
64         offset = 0;
65         *pos += chars;
66         read += chars;           // 累计读入字节数。
67         count -= chars;
68         while (chars-->0)
69             put_fs_byte(*(p++), buf++);
70         brelse(bh);
71     }
72     return read;                 // 返回已读取的字节数，正常退出。
73 }
74

```

12.11 file_dev.c 程序

12.11.1 功能描述

该文件包括 `file_read()` 和 `file_write()` 两个函数。也是供系统调用函数 `read()` 和 `write()` 调用，是用于对普通文件进行读写操作。与上一个文件 `block_dev.c` 类似，该文件也是用于访问文件数据。但是本程序中的函数是通过指定文件路径名方式进行操作。函数参数中给出的是文件 `i` 节点和文件结构信息，通过 `i` 节点中的信息来获取相应的设备号，由 `file` 结构，我们可以获得文件当前的读写指针位置。而上一个文件中的函数则是直接在参数中指定了设备号和文件中的读写位置，是专门用于对块设备文件进行操作的，例如 `/dev/fd0` 设备文件。

12.11.2 代码注释

程序 12-10 linux/fs/file_dev.c

```

1  /*
2  *  linux/fs/file_dev.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <errno.h>           // 错误号头文件。包含系统中各种出错号。
8  #include <fcntl.h>          // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
9
10 #include <linux/sched.h>     // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据等。
11 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <asm/segment.h>     // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
13
14 #define MIN(a,b) (((a)<(b))?(a):(b))    // 取 a,b 中的最小值。
15 #define MAX(a,b) (((a)>(b))?(a):(b))    // 取 a,b 中的最大值。
16
17 // 文件读函数 - 根据 i 节点和文件结构，读取文件中数据。
18 // 由 i 节点我们可以知道设备号，由 filp 结构可以知道文件中当前读写指针位置。buf 指定用
19 // 户空间中缓冲区的位置，count 是需要读取的字节数。返回值是实际读取的字节数，或出错
20 // 号（小于 0）。

```

```

17 int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18 {
19     int left, chars, nr;
20     struct buffer_head * bh;
21
22     // 首先判断参数的有效性。若需要读取的字节计数 count 小于等于零，则返回 0。若还需要读
23     // 取的字节数不等于 0，就循环执行下面操作，直到数据全部读出或遇到问题。在读循环操作
24     // 过程中，我们根据 i 节点和文件表结构信息，并利用 bmap() 得到包含文件当前读写位置的
25     // 数据块在设备上对应的逻辑块号 nr。若 nr 不为 0，则从 i 节点指定的设备上读取该逻辑块。
26     // 如果读操作失败则退出循环。若 nr 为 0，表示指定的数据块不存在，置缓冲块指针为 NULL。
27     // (filp->f_pos)/BLOCK_SIZE 用于计算出文件当前指针所在数据块号。
28     if ((left=count)<=0)
29         return 0;
30     while (left) {
31         if (nr = bmap(inode, (filp->f_pos)/BLOCK_SIZE)) { // inode.c 第 140 行。
32             if (!(bh=bread(inode->i_dev, nr)))
33                 break;
34             } else
35                 bh = NULL;
36
37     // 接着我们计算文件读写指针在数据块中的偏移值 nr，则在该数据块中我们希望读取的字节数
38     // 为 (BLOCK_SIZE - nr)。然后和现在还需读取的字节数 left 作比较，其中小值即为本次操作
39     // 需读取的字节数 chars。如果 (BLOCK_SIZE - nr) > left，则说明该块是需要读取的最后一
40     // 块数据，反之则还需要读取下一块数据。之后调整读写文件指针。指针前移此次将读取的字
41     // 节数 chars。剩余字节计数 left 相应减去 chars。
42     nr = filp->f_pos % BLOCK_SIZE;
43     chars = MIN( BLOCK_SIZE-nr , left );
44     filp->f_pos += chars;
45     left -= chars;
46
47     // 若上面从设备上读到了数据，则将 p 指向缓冲块中开始读取数据的位置，并且复制 chars 字节
48     // 到用户缓冲区 buf 中。否则往用户缓冲区中填入 chars 个 0 值字节。
49     if (bh) {
50         char * p = nr + bh->b_data;
51         while (chars-->0)
52             put_fs_byte(*(p++), buf++);
53         brelse(bh);
54     } else {
55         while (chars-->0)
56             put_fs_byte(0, buf++);
57     }
58
59     // 修改该 i 节点的访问时间为当前时间。返回读取的字节数，若读取字节数为 0，则返回出错号。
60     // CURRENT_TIME 是定义在 include/linux/sched.h 第 142 行上的宏，用于计算 UNIX 时间。即从
61     // 1970 年 1 月 1 日 0 时 0 秒开始，到当前的时间。单位是秒。
62     inode->i_atime = CURRENT_TIME;
63     return (count-left)?(count-left):-ERROR;
64 }
65
66 // 文件写函数 - 根据 i 节点和文件结构信息，将用户数据写入文件中。
67 // 由 i 节点我们可以知道设备号，而由 file 结构可以知道文件中当前读写指针位置。buf 指定
68 // 用户态中缓冲区的位置，count 为需要写入的字节数。返回值是实际写入的字节数，或出错
69 // 号 (小于 0)。
70 int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
71 {

```

```

50     off_t pos;
51     int block, c;
52     struct buffer head * bh;
53     char * p;
54     int i=0;
55
56     /*
57     * ok, append may not work when many processes are writing at the same time
58     * but so what. That way leads to madness anyway.
59     */
60     /*
61     * ok, 当许多进程同时写时, append 操作可能不行, 但那又怎样。不管怎样那样做会
62     * 导致混乱一团。
63     */
64     // 首先确定数据写入文件的位置。如果是要向文件后添加数据, 则将文件读写指针移到文件尾
65     // 部。否则就将在文件当前读写指针处写入。
66     if (filp->f_flags & O_APPEND)
67         pos = inode->i_size;
68     else
69         pos = filp->f_pos;
70     // 然后在已写入字节数 i (刚开始时为 0) 小于指定写入字节数 count 时, 循环执行以下操作。
71     // 在循环操作过程中, 我们先取文件数据块号 ( pos/BLOCK_SIZE ) 在设备上对应的逻辑块号
72     // block。如果对应的逻辑块不存在就创建一块。如果得到的逻辑块号 = 0, 则表示创建失败,
73     // 于是退出循环。否则我们根据该逻辑块号读取设备上的相应逻辑块, 若出错也退出循环。
74     while (i < count) {
75         if (!(block = create_block(inode, pos/BLOCK_SIZE)))
76             break;
77         if (!(bh=bread(inode->i_dev, block)))
78             break;
79         // 此时缓冲块指针 bh 正指向刚读入的文件数据块。现在再求出文件当前读写指针在该数据块中
80         // 的偏移值 c, 并将指针 p 指向缓冲块中开始写入数据的位置, 并置该缓冲块已修改标志。对于
81         // 块中当前指针, 从开始读写位置到块末共可写入 c = (BLOCK_SIZE - c) 个字节。若 c 大于剩余
82         // 还需写入的字节数 (count - i), 则此次只需再写入 c = (count - i) 个字节即可。
83         c = pos % BLOCK_SIZE;
84         p = c + bh->b_data;
85         bh->b_dirt = 1;
86         c = BLOCK_SIZE - c;
87         if (c > count - i) c = count - i;
88         // 在写入数据之前, 我们先预先设置好下一次循环操作要读写文件中的位置。因此我们把 pos
89         // 指针前移此次需写入的字节数。如果此时 pos 位置值超过了文件当前长度, 则修改 i 节点中
90         // 文件长度字段, 并置 i 节点已修改标志。然后把此次要写入的字节数 c 累加到已写入字节计
91         // 数值 i 中, 供循环判断使用。接着从用户缓冲区 buf 中复制 c 个字节到高速缓冲块中 p 指向
92         // 的开始位置处。复制完后就释放该缓冲块。
93         pos += c;
94         if (pos > inode->i_size) {
95             inode->i_size = pos;
96             inode->i_dirt = 1;
97         }
98         i += c;
99         while (c-->0)
100             *(p++) = get_fs_byte(buf++);
101         brelse(bh);
102     }

```

```

// 当数据已经全部写入文件或者在写操作过程中发生问题时就会退出循环。此时我们更改文件
// 修改时间为当前时间，并调整文件读写指针。如果此次操作不是在文件尾添加数据，则把文
// 件读写指针调整到当前读写位置 pos 处，并更改文件 i 节点的修改时间为当前时间。最后返
// 回写入的字节数，若写入字节数为 0，则返回出错号-1。
84     inode->i_mtime = CURRENT_TIME;
85     if (!(filp->f_flags & O_APPEND)) {
86         filp->f_pos = pos;
87         inode->i_ctime = CURRENT_TIME;
88     }
89     return (i?i:-1);
90 }
91

```

12.12 pipe.c 程序

12.12.1 功能描述

管道操作是进程间通信的最基本方式。本程序包括管道文件读写操作函数 `read_pipe()` 和 `write_pipe()`，同时实现了管道系统调用 `sys_pipe()`。这两个函数也是系统调用 `read()` 和 `write()` 的低层实现函数，也仅在 `read_write.c` 中使用。

在创建并初始化管道时，程序会专门申请一个管道 `i` 节点，并为管道分配一页缓冲区（4KB）。管道 `i` 节点的 `i_size` 字段中被设置为指向管道缓冲区的指针，管道数据头部指针存放在 `i_zone[0]` 字段中，而管道数据尾部指针存放在 `i_zone[1]` 字段中。对于读管道操作，数据是从管道尾读出，并使管道尾指针前移读取字节数个位置；对于往管道中的写入操作，数据是向管道头部写入，并使管道头指针前移写入字节数个位置。参见下面的管道示意图 12-28 所示。

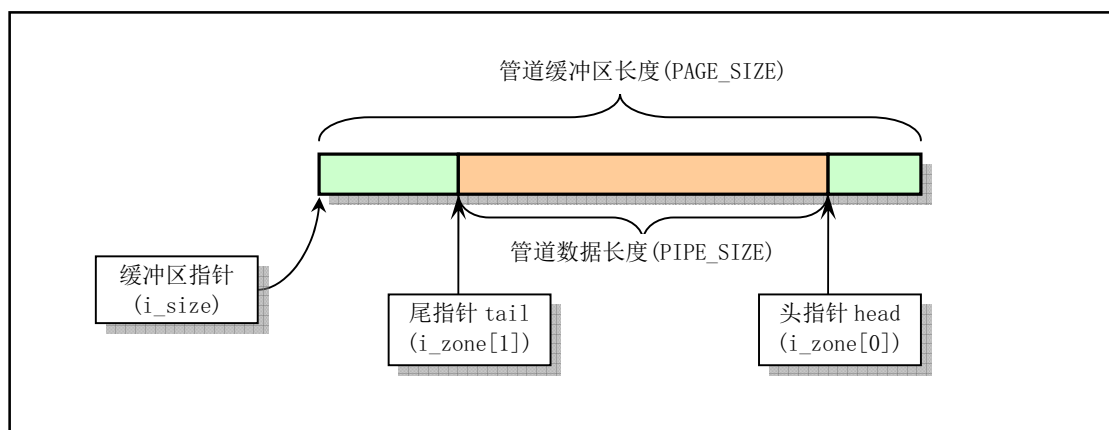


图 12-28 管道缓冲区操作示意图

`read_pipe()` 用于读管道中的数据。若管道中没有数据，就唤醒写管道的进程，而自己则进入睡眠状态。若读到了数据，就相应地调整管道头指针，并把数据传到用户缓冲区中。当把管道中所有的数据都取走后，也要唤醒等待写管道的进程，并返回已读数据字节数。当管道写进程已退出管道操作时，函数就立刻退出，并返回已读的字节数。

`write_pipe()` 函数的操作与读管道函数类似。

系统调用 `sys_pipe()` 用于创建无名管道。它首先在系统的文件表中取得两个表项，然后在当前进程的文件描述符表中也同样寻找两个未使用的描述符表项，用来保存相应的文件结构指针。接着在系统中申请一个空闲 `i` 节点，同时获得管道使用的一个缓冲块。然后对相应的文件结构进行初始化，将一个文件结构设置为只读模式，另一个设置为只写模式。最后将两个文件描述符传给用户。

另外，以上函数中使用的几个与管道操作有关的宏（例如 `PIPE_HEAD()`、`PIPE_TAIL()` 等）定义在 `include/linux/fs.h` 文件第 57--64 行上。

12.12.2 代码注释

程序 12-11 linux/fs/pipe.c

```

1  /*
2  *  linux/fs/pipe.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <signal.h>          // 信号头文件。定义信号符号常量，信号结构及操作函数原型。
8
9  #include <linux/sched.h>     // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
10 #include <linux/mm.h>        /* for get_free_page */    /* 使用其中的 get_free_page */
11                             // 内存管理头文件。含有页面长度定义和一些页面释放函数原型。
12 #include <asm/segment.h>     // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
13
14     ///// 管道读操作函数。
15     // 参数 inode 是管道对应的 i 节点，buf 是用户数据缓冲区指针，count 是读取的字节数。
16 int read_pipe(struct m_inode * inode, char * buf, int count)
17 {
18     int chars, size, read = 0;
19
20     // 如果需要读取的字节计数 count 大于 0，我们就循环执行以下操作。在循环读操作过程中，
21     // 若当前管道中没有数据（size=0），则唤醒等待该节点的进程，这通常是写管道进程。如果
22     // 已没有写管道者，即 i 节点引用计数值小于 2，则返回已读字节数退出。否则在该 i 节点上
23     // 睡眠，等待信息。宏 PIPE_SIZE 定义在 include/linux/fs.h 中。
24     while (count>0) {
25         while (!(size=PIPE_SIZE(*inode))) {           // 取管道中数据长度值。
26             wake_up(&inode->i_wait);
27             if (inode->i_count != 2)    /* are there any writers? */
28                 return read;
29             sleep_on(&inode->i_wait);
30         }
31         // 此时说明管道（缓冲区）中有数据。于是我们取管道尾指针到缓冲区末端的字节数 chars。
32         // 如果其大于还需要读取的字节数 count，则令其等于 count。如果 chars 大于当前管道中含
33         // 有数据的长度 size，则令其等于 size。然后把需读字节数 count 减去此次可读的字节数
34         // chars，并累加已读字节数 read。
35         chars = PAGE_SIZE-PIPE_TAIL(*inode);
36         if (chars > count)
37             chars = count;
38         if (chars > size)
39             chars = size;
40         count -= chars;
41         read += chars;
42     }
43     // 再令 size 指向管道尾指针处，并调整当前管道尾指针（前移 chars 字节）。若尾指针超过

```

```

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

```

```

65         ((char *)inode->i_size)[size++]=get\_fs\_byte(buf++);
66     }
    // 当此次写管道操作结束，则唤醒等待管道的进程，返回已写入的字节数，退出。
67     wake\_up(&inode->i_wait);
68     return written;
69 }
70
    ///// 创建管道系统调用。
    // 在 fildes 所指的数组中创建一对文件句柄（描述符）。这对文件句柄指向一管道 i 节点。
    // 参数：fildes - 文件句柄数组。fildes[0]用于读管道数据，fildes[1]向管道写入数据。
    // 成功时返回 0，出错时返回-1。
71 int sys\_pipe(unsigned long * fildes)
72 {
73     struct m\_inode * inode;
74     struct file * f[2];           // 文件结构数组。
75     int fd[2];                   // 文件句柄数组。
76     int i, j;
77
    // 首先从系统文件表中取两个空闲项（引用计数字段为 0 的项），并分别设置引用计数为 1。
    // 若只有 1 个空闲项，则释放该项（引用计数复位）。若没有找到两个空闲项，则返回 -1。
78     j=0;
79     for(i=0; j<2 && i<NR\_FILE; i++)
80         if (!file\_table[i].f_count)
81             (f[j++] = i + file\_table) ->f_count++;
82     if (j==1)
83         f[0]->f_count=0;
84     if (j<2)
85         return -1;
    // 针对上面取得的两个文件表结构项，分别分配一文件句柄号，并使进程文件结构指针数组的
    // 两项分别指向这两个文件结构。而文件句柄即是该数组的索引号。类似地，如果只有一个空
    // 闲文件句柄，则释放该句柄（置空相应数组项）。如果没有找到两个空闲句柄，则释放上面
    // 获取的两个文件结构项（复位引用计数值），并返回 -1。
86     j=0;
87     for(i=0; j<2 && i<NR\_OPEN; i++)
88         if (!current->filp[i]) {
89             current->filp[fd[j]=i] = f[j];
90             j++;
91         }
92     if (j==1)
93         current->filp[fd[0]] = NULL;
94     if (j<2) {
95         f[0]->f_count=f[1]->f_count=0;
96         return -1;
97     }
    // 然后利用函数 get\_pipe\_inode() 申请一个管道使用的 i 节点，并为管道分配一页内存作为缓
    // 冲区。如果不成功，则相应释放两个文件句柄和文件结构项，并返回-1。
98     if (!(inode=get\_pipe\_inode())) {           // fs/inode.c, 第 228 行开始处。
99         current->filp[fd[0]] =
100             current->filp[fd[1]] = NULL;
101         f[0]->f_count = f[1]->f_count = 0;
102         return -1;
103     }
    // 如果管道 i 节点申请成功，则对两个文件结构进行初始化操作，让它们都指向同一个管道 i 节

```

```

// 点，并把读写指针都置零。第 1 个文件结构的文件模式置为读，第 2 个文件结构的文件模式置
// 为写。最后将文件句柄数组复制到对应的用户空间数组中，成功返回 0，退出。
104     f[0]->f_inode = f[1]->f_inode = inode;
105     f[0]->f_pos = f[1]->f_pos = 0;
106     f[0]->f_mode = 1;                      /* read */
107     f[1]->f_mode = 2;                      /* write */
108     put_fs_long(fd[0], 0+fildes);
109     put_fs_long(fd[1], 1+fildes);
110     return 0;
111 }
112

```

12.13 char_dev.c 程序

12.13.1 功能描述

char_dev.c 文件包括字符设备文件访问函数。主要有 rw_ttyx()、rw_tty()、rw_memory()和 rw_char()。另外还有一个设备读写函数指针表。该表的项号代表主设备号。

rw_ttyx()是串口终端设备读写函数，其主设备号是 4。通过调用 tty 的驱动程序实现了对串口终端的读写操作。

rw_tty()是控制台终端读写函数，主设备号是 5。实现原理与 rw_ttyx()相同，只是对进程能否进行控制台操作有所限制。

rw_memory()是内存设备文件读写函数，主设备号是 1。实现了对内存映像的字节操作。但 linux 0.11 版内核对次设备号是 0、1、2 的操作还没有实现。直到 0.96 版才开始实现次设备号 1 和 2 的读写操作。

rw_char()是字符设备读写操作的接口函数。其他字符设备通过该函数对字符设备读写函数指针表进行相应字符设备的操作。文件系统的操作函数 open()、read()等都通过它对所有字符设备文件进行操作。

12.13.2 代码注释

程序 12-12 linux/fs/char_dev.c

```

1  /*
2  *  linux/fs/char_dev.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <errno.h>          // 错误号头文件。包含系统中各种出错号。
8  #include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。
9
10 #include <linux/sched.h>    // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
11 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
12
13 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14 #include <asm/io.h>        // io 头文件。定义硬件端口输入/输出宏汇编语句。
15

```



```

16 extern int tty_read(unsigned minor, char * buf, int count); // 终端读。
17 extern int tty_write(unsigned minor, char * buf, int count); // 终端写。
18
19 // 定义字符设备读写函数指针类型。
20 typedef (*crw_ptr)(int rw, unsigned minor, char * buf, int count, off_t * pos);
21
22 // 串口终端读写操作函数。
23 // 参数: rw - 读写命令; minor - 终端子设备号; buf - 缓冲区; cout - 读写字节数;
24 // pos - 读写操作当前指针, 对于终端操作, 该指针无用。
25 // 返回: 实际读写的字节数。若失败则返回出错码。
26 static int rw_ttyx(int rw, unsigned minor, char * buf, int count, off_t * pos)
27 {
28     return ((rw==READ)?tty_read(minor, buf, count):
29             tty_write(minor, buf, count));
30 }
31
32 // 终端读写操作函数。
33 // 同上 rw_ttyx(), 只是增加了对进程是否有控制终端的检测。
34 static int rw_tty(int rw, unsigned minor, char * buf, int count, off_t * pos)
35 {
36     // 若进程没有对应的控制终端, 则返回出错号。否则调用终端读写函数 rw_ttyx(), 并返回
37     // 实际读写字节数。
38     if (current->tty<0)
39         return -EPERM;
40     return rw_ttyx(rw, current->tty, buf, count, pos);
41 }
42
43 // 内存数据读写。未实现。
44 static int rw_ram(int rw, char * buf, int count, off_t * pos)
45 {
46     return -EIO;
47 }
48
49 // 物理内存数据读写操作函数。未实现。
50 static int rw_mem(int rw, char * buf, int count, off_t * pos)
51 {
52     return -EIO;
53 }
54
55 // 内核虚拟内存数据读写函数。未实现。
56 static int rw_kmem(int rw, char * buf, int count, off_t * pos)
57 {
58     return -EIO;
59 }
60
61 // 端口读写操作函数。
62 // 参数: rw - 读写命令; buf - 缓冲区; cout - 读写字节数; pos - 端口地址。
63 // 返回: 实际读写的字节数。
64 static int rw_port(int rw, char * buf, int count, off_t * pos)
65 {
66     int i=*pos;
67
68     // 对于所要求读写的字节数, 并且端口地址小于 64k 时, 循环执行单个字节的读写操作。

```

```

// 若是读命令，则从端口 i 中读取一字节内容并放到用户缓冲区中。若是写命令，则从用
// 户数据缓冲区中取一字节输出到端口 i。
53 while (count-->0 && i<65536) {
54     if (rw==READ)
55         put_fs_byte(inb(i), buf++);
56     else
57         outb(get_fs_byte(buf++), i);
58     i++; // 前移一个端口。[??]
59 }
// 然后计算读/写的字节数，调整相应读写指针，并返回读/写的字节数。
60 i -= *pos;
61 *pos += i;
62 return i;
63 }
64
//// 内存读写操作函数。
65 static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos)
66 {
// 根据内存设备子设备号，分别调用不同的内存读写函数。
67     switch(minor) {
68         case 0:
69             return rw_ram(rw, buf, count, pos);
70         case 1:
71             return rw_mem(rw, buf, count, pos);
72         case 2:
73             return rw_kmem(rw, buf, count, pos);
74         case 3:
75             return (rw==READ)?0:count; /* rw_null */
76         case 4:
77             return rw_port(rw, buf, count, pos);
78         default:
79             return -EIO;
80     }
81 }
82
// 定义系统中设备种数。
83 #define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))
84
// 字符设备读写函数指针表。
85 static crw_ptr crw_table[]={
86     NULL, /* nodev */ /* 无设备(空设备) */
87     rw_memory, /* /dev/mem etc */ /* /dev/mem 等 */
88     NULL, /* /dev/fd */ /* /dev/fd 软驱 */
89     NULL, /* /dev/hd */ /* /dev/hd 硬盘 */
90     rw_ttyx, /* /dev/ttyx */ /* /dev/ttyx 串口终端 */
91     rw_tty, /* /dev/tty */ /* /dev/tty 终端 */
92     NULL, /* /dev/lp */ /* /dev/lp 打印机 */
93     NULL; /* unnamed pipes */ /* 未命名管道 */
94
//// 字符设备读写操作函数。
// 参数: rw -读写命令; dev -设备号; buf -缓冲区; count -读写字节数; pos -读写指针。
// 返回: 实际读/写字节数。
95 int rw_char(int rw, int dev, char * buf, int count, off_t * pos)

```

```

96 {
97     crw\_ptr call_addr;
98     // 如果设备号超出系统设备数，则返回出错码。如果该设备没有对应的读/写函数，也返回出
    // 错码。否则调用对应设备的读写操作函数，并返回实际读/写的字节数。
99     if (MAJOR(dev) >= NRDEVS)
100         return -ENODEV;
101     if (!(call_addr = crw\_table[MAJOR(dev)]))
102         return -ENODEV;
103     return call_addr(rw, MINOR(dev), buf, count, pos);
104 }
105

```

12.14 read_write.c 程序

12.14.1 功能描述

该文件实现了文件操作系统调用 `read()`、`write()` 和 `lseek()`。`read()` 和 `write()` 将根据不同的文件类型，分别调用前面 4 个文件中实现的相应读写函数。因此本文件是前面 4 个文件中函数的上层接口实现。`lseek()` 用于设置文件读写指针。

`read()` 系统调用首先判断所给参数的有效性，然后根据文件的 `i` 节点信息判断文件的类型。若是管道文件则调用程序 `pipe.c` 中的读函数；若是字符设备文件，则调用 `char_dev.c` 中的 `rw_char()` 字符读函数；如果是块设备文件，则执行 `block_dev.c` 程序中的块设备读操作，并返回读取的字节数；如果是目录文件或一般正规文件，则调用 `file_dev.c` 中的文件读函数 `file_read()`。`write()` 系统调用的实现与 `read()` 类似。

`lseek()` 系统调用将对文件句柄对应文件结构中的当前读写指针进行修改。对于读写指针不能移动的文件和管道文件，将给出错误号，并立即返回。

12.14.2 代码注释

程序 12-13 linux/fs/read_write.c

```

1  /*
2   *  linux/fs/read_write.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <sys/stat.h>    // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
8  #include <errno.h>      // 错误号头文件。包含系统中各种出错号。
9  #include <sys/types.h>  // 类型头文件。定义了基本的系统数据类型。
10
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
13 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14
    // 字符设备读写函数。fs/char_dev.c，第 95 行。
15 extern int rw\_char(int rw, int dev, char * buf, int count, off\_t * pos);
    // 读管道操作函数。fs/pipe.c，第 13 行。
16 extern int read\_pipe(struct m\_inode * inode, char * buf, int count);

```

```

// 写管道操作函数。fs/pipe.c, 第 41 行。
17 extern int write_pipe(struct m_inode * inode, char * buf, int count);
// 块设备读操作函数。fs/block_dev.c, 第 47 行。
18 extern int block_read(int dev, off_t * pos, char * buf, int count);
// 块设备写操作函数。fs/block_dev.c, 第 14 行。
19 extern int block_write(int dev, off_t * pos, char * buf, int count);
// 读文件操作函数。fs/file_dev.c, 第 17 行。
20 extern int file_read(struct m_inode * inode, struct file * filp,
21                     char * buf, int count);
// 写文件操作函数。fs/file_dev.c, 第 48 行。
22 extern int file_write(struct m_inode * inode, struct file * filp,
23                      char * buf, int count);
24
///// 重定位文件读写指针系统调用。
// 参数 fd 是文件句柄, offset 是新的文件读写指针偏移值, origin 是偏移的起始位置, 可有
// 三种选择: SEEK_SET (0, 从文件开始处)、SEEK_CUR (1, 从当前读写位置)、SEEK_END (
// 2, 从文件尾处)。
25 int sys_lseek(unsigned int fd, off_t offset, int origin)
26 {
27     struct file * file;
28     int tmp;
29
// 首先判断函数提供的参数有效性。如果文件句柄值大于程序最多打开文件数 NR_OPEN(20),
// 或者该句柄的文件结构指针为空, 或者对应文件结构的 i 节点字段为空, 或者指定设备文件
// 指针是不可定位的, 则返回出错码并退出。如果文件对应的 i 节点是管道节点, 则返回出错
// 码退出。因为管道头尾指针不可随意移动!
30     if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode)
31         || !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
32         return -EBADF;
33     if (file->f_inode->i_pipe)
34         return -ESPIPE;
// 然后根据设置的定位标志, 分别重新定位文件读写指针。
35     switch (origin) {
// origin = SEEK_SET, 要求以文件起始处作为原点设置文件读写指针。若偏移值小于零, 则出
// 错返回错误码。否则设置文件读写指针等于 offset。
36         case 0:
37             if (offset < 0) return -EINVAL;
38             file->f_pos=offset;
39             break;
// origin = SEEK_CUR, 要求以文件当前读写指针处作为原点重定位读写指针。如果文件当前指
// 针加上偏移值小于 0, 则返回出错码退出。否则在当前读写指针上加上偏移值。
40         case 1:
41             if (file->f_pos+offset < 0) return -EINVAL;
42             file->f_pos += offset;
43             break;
// origin = SEEK_END, 要求以文件末尾作为原点重定位读写指针。此时若文件大小加上偏移值
// 小于零则返回出错码退出。否则重定位读写指针为文件长度加上偏移值。
44         case 2:
45             if ((tmp=file->f_inode->i_size+offset) < 0)
46                 return -EINVAL;
47             file->f_pos = tmp;
48             break;
// 若 origin 设置无效, 返回出错码退出。

```

```

49         default:
50             return -EINVAL;
51     }
52     return file->f_pos;           // 最后返回重定位后的文件读写指针值。
53 }
54
55 // 读文件系统调用。
56 // 参数 fd 是文件句柄, buf 是缓冲区, count 是欲读字节数。
57 int sys_read(unsigned int fd, char * buf, int count)
58 {
59     struct file * file;
60     struct m_inode * inode;
61
62     // 函数首先对参数有效性进行判断。如果文件句柄值大于程序最多打开文件数 NR_OPEN, 或者
63     // 需要读取的字节计数值小于 0, 或者该句柄的文件结构指针为空, 则返回出错码并退出。若
64     // 需读取的字节数 count 等于 0, 则返回 0 退出
65     if (fd >= NR_OPEN || count < 0 || !(file = current->filp[fd]))
66         return -EINVAL;
67     if (!count)
68         return 0;
69     // 然后验证存放数据的缓冲区内存限制。并取文件的 i 节点。用于根据该 i 节点的属性, 分别
70     // 调用相应的读操作函数。若是管道文件, 并且是读管道文件模式, 则进行读管道操作, 若成
71     // 功则返回读取的字节数, 否则返回出错码, 退出。如果是字符型文件, 则进行读字符设备操
72     // 作, 并返回读取的字符数。如果是块设备文件, 则执行块设备读操作, 并返回读取的字节数。
73     verify_area(buf, count);
74     inode = file->f_inode;
75     if (inode->i_pipe)
76         return (file->f_mode & 1) ? read_pipe(inode, buf, count) : -EIO;
77     if (S_ISCHR(inode->i_mode))
78         return rw_char(READ, inode->i_zone[0], buf, count, &file->f_pos);
79     if (S_ISBLK(inode->i_mode))
80         return block_read(inode->i_zone[0], &file->f_pos, buf, count);
81     // 如果是目录文件或者是常规文件, 则首先验证读取字节数 count 的有效性并进行调整 (若读
82     // 取字节数加上文件当前读写指针值大于文件长度, 则重新设置读取字节数为 文件长度-当前
83     // 读写指针值, 若读取数等于 0, 则返回 0 退出), 然后执行文件读操作, 返回读取的字节数
84     // 并退出。
85     if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
86         if (count + file->f_pos > inode->i_size)
87             count = inode->i_size - file->f_pos;
88         if (count <= 0)
89             return 0;
90         return file_read(inode, file, buf, count);
91     }
92     // 执行到这里, 说明我们无法判断文件的属性。则打印节点文件属性, 并返回出错码退出。
93     printk("(Read) inode->i_mode=%06o\n|r", inode->i_mode);
94     return -EINVAL;
95 }
96
97 // 写文件系统调用。
98 // 参数 fd 是文件句柄, buf 是用户缓冲区, count 是欲写字节数。
99 int sys_write(unsigned int fd, char * buf, int count)
100 {
101     struct file * file;

```

```

86     struct m_inode * inode;
87
88     // 同样地，我们首先判断函数参数的有效性。如果进程文件句柄值大于程序最多打开文件数
89     // NR_OPEN，或者需要写入的字节计数小于 0，或者该句柄的文件结构指针为空，则返回出错
90     // 码并退出。如果需读取的字节数 count 等于 0，则返回 0 退出
91     if (fd>=NR_OPEN || count < 0 || !(file=current->filp[fd]))
92         return -EINVAL;
93     if (!count)
94         return 0;
95     // 然后验证存放数据的缓冲区内存限制。并取文件的 i 节点。用于根据该 i 节点的属性，分别
96     // 调用相应的读操作函数。若是管道文件，并且是写管道文件模式，则进行写管道操作，若成
97     // 功则返回写入的字节数，否则返回出错码退出。如果是字符设备文件，则进行写字符设备操
98     // 作，返回写入的字符数退出。如果是块设备文件，则进行块设备写操作，并返回写入的字节
99     // 数退出。若是常规文件，则执行文件写操作，并返回写入的字节数，退出。
100     inode=file->f_inode;
101     if (inode->i_pipe)
102         return (file->f_mode&2)?write_pipe(inode, buf, count):-EIO;
103     if (S_ISCHR(inode->i_mode))
104         return rw_char(WRITE, inode->i_zone[0], buf, count, &file->f_pos);
105     if (S_ISBLK(inode->i_mode))
106         return block_write(inode->i_zone[0], &file->f_pos, buf, count);
107     if (S_ISREG(inode->i_mode))
108         return file_write(inode, file, buf, count);
109     // 执行到这里，说明我们无法判断文件的属性。则打印节点文件属性，并返回出错码退出。
110     printk("(Write)inode->i_mode=%06o\n\r", inode->i_mode);
111     return -EINVAL;
112 }
113
114

```

12.14.3 用户程序读写操作过程

在看完上面程序后，我们应该可以清楚地理解一个用户程序中的读写操作是如何执行的。下面我们以内核中的读操作函数为例具体说明用户程序中的一个读文件函数调用是如何执行并完成的。

通常，应用程序不直接调用 Linux 的系统调用（System Calls），而是通过调用函数库（例如 libc.a）中的子程序进行操作的。但是若为了提高一些效率，当然也是可以直接进行调用的。对于一个基本的函数库来讲，通常需要提供以下一些基本函数或子程序的集合：

- ◆ 系统调用接口函数
- ◆ 内存分配管理函数
- ◆ 信号处理函数集
- ◆ 字符串处理函数
- ◆ 标准输入输出函数
- ◆ 其他函数集，如 `bsd` 函数、加解密函数、算术运算函数、终端操作函数和网络套接字函数集等。

在这些函数集中，系统调用函数是操作系统的底层接口函数。许多牵涉到系统调用的函数都会调用系统调用接口函数集中具有标准名称的系统函数，而不是直接使用 Linux 的系统终端调用接口。这样做可以很大程度上让一个函数库与其所在的操作系统无关，让函数库有较高的可移植性。对于一个新的函数库源代码，只要将其中涉及系统调用的部分（系统接口部分）替换成新操作系统的系统调用，就基本上能完成该函数库的移植工作。

库中的子程序可以看作是应用程序与内核系统之间的中间层，它的主要作用除了提供一些不属于内核的计算函数等功能函数外，还为应用程序执行系统调用提供“包裹函数”。这样做一来可以简化调用接口，是接口更简单容易记忆，二来可以在这些包裹函数中进行一些参数验证，出错处理，因此能使得程

序更加可靠稳定。

对于 Linux 系统，所有输入输出都是通过读写文件完成的。因为所有的外围设备都是以文件形式在系统中呈现，这样使用统一的文件句柄就可以处理程序与外设之间的所有访问操作。在通常情况下，在读写一个文件之前我们需要首先使用打开文件（`open file`）操作来通知操作系统将要开始的行动。如果想在文件上执行写操作，那么首先可能需要先创建这个文件或者将文件中以前的内容删除。操作系统还需要检查你是否有权来执行这些操作。如果一切正常的话，打开操作会向程序返回一个文件描述符（`file escriptor`），文件描述符将替代文件名来确定所访问的文件，它与 MS-DOS 中文件句柄（`file handle`）作用一样。此时一个打开着的文件的所有信息都由系统来维护，用户程序只需要使用文件描述符来访问文件。

文件读写分别使用 `read` 和 `write` 系统调用，用户程序一般通过访问函数库中的 `read` 和 `write` 函数来执行这两个系统调用。这两个函数的定义如下：

```
int read(int fd, char *buf, int n);
int write(int fd, char *buf, int n);
```

这两个函数的头一个参数是文件描述符。第二个参数是一个字符缓冲阵列，用于存放读取或被写出的数据。第三个参数是需要读写的数据字节数。函数返回值是一次调用时传输的字节计数值。对于读文件操作，返回的值可能会比想要读的数据小。如果返回值是 0，则表示已经读到文件尾。如果返回-1，则表示读操作遇到错误。对于写操作，返回的值是实际写入的字节数，如果该值与第三个参数指定的值不等，这表示写操作遇到了错误。对于读函数，它在函数库中的实现形式如下：

```
#define __LIBRARY__
#include <unistd.h>

_syscall3(int, read, int, fd, char *, buf, off_t, count)
```

其中 `_syscall3()` 是一个宏，定义在 `unistd.h` 头文件第 172 行开始处。若将该宏以上面的具体参数展开，我们可得到以下代码：

```
int read(int fd, char *buf, off_t count)
{
    long __res;
    __asm__ volatile (
        "int $0x80"
        : "=a" (__res)
        : "" (__NR_read), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(count)));
    if (__res >= 0)
        return int __res;
    errno = -__res;
    return -1;
}
```

可以看出，这个展开的宏就是一个读操作函数的具体实现。从此处程序进入系统内核中执行。其中使用了嵌入汇编语句以功能号 `__NR_read` (3) 执行了 Linux 的系统中断调用 `0x80`。该中断调用在 `eax` (`__res`) 寄存器中返回了实际读取的字节数。若返回的值小于 0，则表示此次读操作出错，于是将出错号取反后存入全局变量 `errno` 中，并向调用程序返回-1 值。

在 Linux 内核中，读操作在文件系统的 `read_write.c` 文件中实现。当执行了上述系统中断调用时，在

该系统中断程序中就会去调用执行 read_write.c 文件中第 55 行开始的 sys_read() 函数。sys_read() 函数的原型定义如下：

```
int sys_read(unsigned int fd, char *buf, int count)
```

该函数首先判断参数的有效性。如果文件描述符值大于系统最多同时打开的最大文件数，或者需要读取的字节数值小于 0，或者该文件还没有执行过打开操作（此时文件描述符所索引的文件结构项指针为空），这返回一个负的出错代码。接着内核程序验证将要存放读取数据的缓冲区大小是否合适。在验证过程中，内核程序会根据指定的读取字节数对缓冲区 buf 的大小进行验证，如果 buf 太小，这系统会对其进行扩充。因此，若用户程序开辟的内存缓冲区太小的话就有可能冲毁后面的数据。

随后内核代码会从文件描述符对应的内部文件表结构中获得该文件的 i 节点结构，并根据节点中的标志信息对该文件进行分类判断，调用下面对应类型的读操作函数，并返回所读取的实际字节数。

- 如果该文件是管道文件，则调用读管道函数 read_pipe()（在 fs/pipe.c 中实现）进行操作。
- 如果是字符设备文件，则调用读字符设备操作函数 rw_char()（在 fs/char_dev.c 中实现）。该函数再会根据具体的字符设备子类型调用字符设备驱动程序或对内存字符设备进行操作。
- 如果是块设备文件，则调用块设备读操作函数 block_read()（在 fs/block_dev.c 中实现）。该函数则调用内存高速缓冲管理程序 fs/buffer.c 中的读块函数 bread()，最后调用到块设备驱动程序中的 ll_rw_block() 函数执行实际的块设备读操作。
- 如果该文件是一般普通常规文件，则调用常规文件读函数 file_read()（在 fs/file_read.c 中实现）进行读数据操作。该函数与读块设备操作类似，最后也会去调用执行文件系统所在块设备的底层驱动程序访问函数 ll_rw_block()，但是 file_read() 还需要维护相关的内部文件表结构中的信息，例如移动文件当前指针。

当读操作的系统调用返回时，函数库中的 read() 函数就可以根据系统调用返回值来判断此次操作是否正确。若返回的值小于 0，则表示此次读操作出错，于是将出错号取反后存入全局变量 errno 中，并向应用程序返回 -1 值。从用户程序执行 read() 函数到进入内核中进行实际操作的整个过程参见图 12-29 所示。

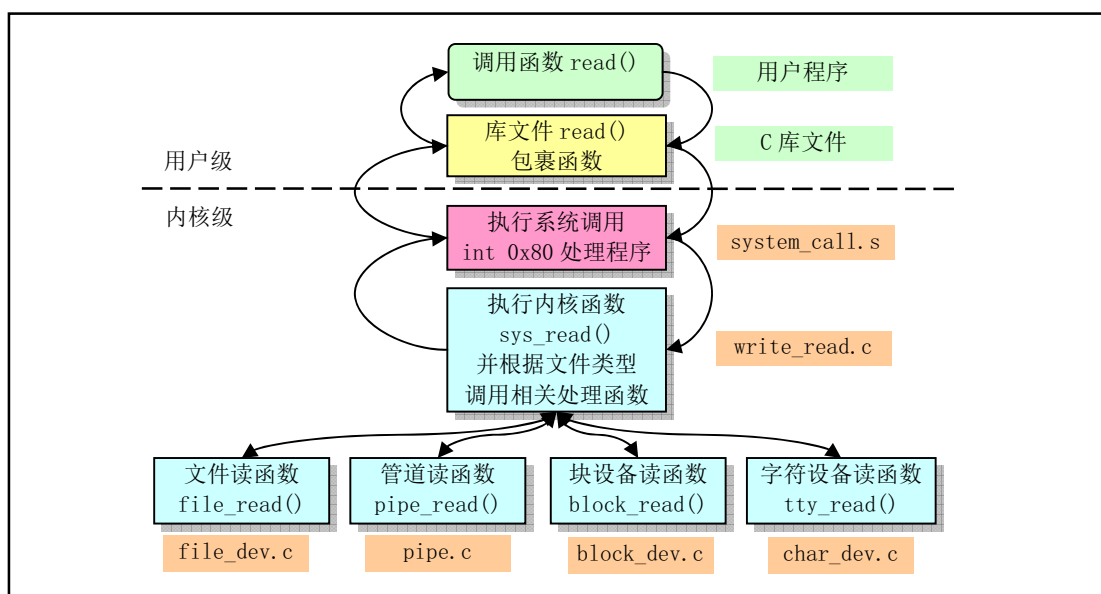


图 12-29 read() 函数调用执行过程

12.15 open.c 程序

从本节开始描述的所有程序均属于文件系统高层操作和管理部分，即本程序的第 4 部分。这部分包括 5 个程序，分别是 open.c、exec.c、stat.c、fcntl.c 和 ioctl.c 程序。

open.c 程序主要包含文件访问操作系统调用；exec.c 主要包含程序加载和执行函数 execve()；stat.c 程序用于取得一个文件的状态信息；fcntl.c 程序实现文件访问控制管理；ioctl.c 程序则用于控制设备的访问操作。

12.15.1 功能描述

本文件实现了许多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。

12.15.2 代码注释

程序 12-14 linux/fs/open.c

```

1  /*
2  *  linux/fs/open.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <string.h>      // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8  #include <errno.h>      // 错误号头文件。包含系统中各种出错号。
9  #include <fcntl.h>      // 文件控制头文件。用于文件及其描述符操作控制常数符号定义。
10 #include <sys/types.h>   // 类型头文件。定义基本的系统和文件系统统计信息结构和类型。
11 #include <utime.h>      // 用户时间头文件。定义访问和修改时间结构以及 utime() 原型。
12 #include <sys/stat.h>    // 文件状态头文件。含有文件状态结构 stat {} 和符号常量等。
13
14 #include <linux/sched.h> // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
15 #include <linux/tty.h>   // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
16 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
17 #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
19 // 取文件系统信息。
20 // 参数 dev 是含有已安装文件系统的设备号。ubuf 是一个 ustat 结构缓冲区指针，用于存放
21 // 系统返回的文件系统信息。该系统调用用于返回已安装 (mounted) 文件系统的统计信息。
22 // 成功时返回 0，并且 ubuf 指向的 ustate 结构被添入文件系统总空闲块数和空闲 i 节点数。
23 // ustat 结构定义在 include/sys/types.h 中。
24
25 int sys_ustat(int dev, struct ustat * ubuf)
26 {
27     return -ENOSYS;          // 出错码：功能还未实现。
28 }
29
30 // 设置文件访问和修改时间。
31 // 参数 filename 是文件名，times 是访问和修改时间结构指针。
32 // 如果 times 指针不为 NULL，则取 utimbuf 结构中的时间信息来设置文件的访问和修改时间。
33 // 如果 times 指针是 NULL，则取系统当前时间来设置指定文件的访问和修改时间域。

```

```

24 int sys utime(char * filename, struct utimbuf * times)
25 {
26     struct m_inode * inode;
27     long actime, modtime;
28     // 文件的时间信息保存在其 i 节点中。因此我们首先根据文件名取得对应 i 节点。如果没有找
    // 到，则返回出错码。
29     if (!(inode=namei(filename)))
30         return -ENOENT;
    // 如果提供的访问和修改时间结构指针 times 不为 NULL，则从结构中读取用户设置的时间值。
    // 否则就用系统当前时间来设置文件的访问和修改时间。
31     if (times) {
32         actime = get_fs_long((unsigned long *) &times->actime);
33         modtime = get_fs_long((unsigned long *) &times->modtime);
34     } else
35         actime = modtime = CURRENT_TIME;
    // 然后修改 i 节点中的访问时间字段和修改时间字段。再设置 i 节点已修改标志，放回该 i 节
    // 点，并返回 0。
36     inode->i_atime = actime;
37     inode->i_mtime = modtime;
38     inode->i_dirt = 1;
39     iput(inode);
40     return 0;
41 }
42
43 /*
44  * XXX should we use the real or effective uid? BSD uses the real uid,
45  * so as to make this call useful to setuid programs.
46  */
/*
 * XXX 我们该用真实用户 id (ruid) 还是有效用户 id (euid) ? BSD 系统使用了
 * 真实用户 id，以使该调用可以供 setuid 程序使用。
 * （注：POSIX 标准建议使用真实用户 ID）。
 * （注 1：英文注释开始的 'XXX' 表示重要提示）。
 */
///// 检查文件的访问权限。
// 参数 filename 是文件名，mode 是检查的访问属性，它有 3 个有效比特位组成：R_OK(值 4)、
// W_OK(2)、X_OK(1) 和 F_OK(0) 组成，分别表示检测文件是否可读、可写、可执行和文件是
// 否存在。如果访问允许的话，则返回 0，否则返回出错码。
47 int sys access(const char * filename, int mode)
48 {
49     struct m_inode * inode;
50     int res, i_mode;
51     // 文件的访问权限信息也同样保存在文件的 i 节点结构中，因此我们要先取得对应文件名的 i
    // 节点。检测的访问属性 mode 由低 3 位组成，因此需要与上八进制 0007 来清除所有高比特位。
    // 如果文件名对应的 i 节点不存在，则返回没有许可权限出错码。若 i 节点存在，则取 i 节点
    // 钟文件属性码，并放回该 i 节点。另外，56 行上语句“iput(inode);”最后放在 60 行之后。
52     mode &= 0007;
53     if (!(inode=namei(filename)))
54         return -EACCES; // 出错码：无访问权限。
55     i_mode = res = inode->i_mode & 0777;
56     iput(inode);

```

```

// 如果当前进程用户是该文件的宿主，则取文件宿主属性。否则如果当前进程用户与该文件宿
// 主同属一组，则取文件组属性。否则，此时 res 最低 3 比特是其他人访问该文件的许可属性。
// [[?? 这里应 res >>3 ??]
57     if (current->uid == inode->i_uid)
58         res >>= 6;
59     else if (current->gid == inode->i_gid)
60         res >>= 6;
// 此时 res 的最低 3 比特是根据当前进程用户与文件的关系选择出来的访问属性位。现在我们
// 来判断这 3 比特。如果文件属性具有参数所查询的属性位 mode，则访问许可，返回 0。
61     if ((res & 0007 & mode) == mode)
62         return 0;
63     /*
64     * XXX we are doing this test last because we really should be
65     * swapping the effective with the real user id (temporarily),
66     * and then calling suser() routine. If we do call the
67     * suser() routine, it needs to be called last.
68     */
69     /*
70     * XXX 我们最后才做下面的测试，因为我们实际上需要交换有效用户 ID 和
71     * 真实用户 ID（临时地），然后才调用 suser() 函数。如果我们确实要调用
72     * suser() 函数，则需要最后才被调用。
73     */
74     // 如果当前用户 ID 为 0（超级用户）并且屏蔽码执行位是 0 或者文件可以被任何人执行、搜
75     // 索，则返回 0。否则返回出错码。
76     if ((!current->uid) &&
77         (!(mode & 1) || (i_mode & 0111)))
78         return 0;
79     return -EACCES;
80 }
81
82 // 改变当前工作目录系统调用。
83 // 参数 filename 是目录名。
84 // 操作成功则返回 0，否则返回出错码。
85 int sys\_chdir(const char * filename)
86 {
87     struct m\_inode * inode;
88
89     // 改变当前工作目录就是要求把进程任务结构的当前工作目录字段指向给定目录名的 i 节点。
90     // 因此我们首先取目录名的 i 节点。如果目录名对应的 i 节点不存在，则返回出错码。如果该
91     // i 节点不是一个目录 i 节点，则放回该 i 节点，并返回出错码。
92     if (!(inode = namei(filename)))
93         return -ENOENT; // 出错码：文件或目录不存在。
94     if (!S\_ISDIR(inode->i_mode)) {
95         iput(inode);
96         return -ENOTDIR; // 出错码：不是目录名。
97     }
98     // 然后释放进程原工作目录 i 节点，并使其指向新设置的工作目录 i 节点。返回 0。
99     iput(current->pwd);
100    current->pwd = inode;
101    return (0);
102 }
103
104 // 改变根目录系统调用。

```

```

// 把指定的目录名设置成为当前进程的根目录'/'。
// 如果操作成功则返回 0，否则返回出错码。
90 int sys_chroot(const char * filename)
91 {
92     struct m_inode * inode;
93
94     // 该调用用于改变当前进程任务结构中的根目录字段 root，让其指向参数给定目录名的 i 节点。
95     // 如果目录名对应的 i 节点不存在，则返回出错码。如果该 i 节点不是目录 i 节点，则放回该
96     // i 节点，也返回出错码。
97     if (!(inode=namei(filename)))
98         return -ENOENT;
99     if (!S_ISDIR(inode->i_mode)) {
100         iput(inode);
101         return -ENOTDIR;
102     }
103 // 然后释放当前进程的根目录 i 节点，并重新设置为指定目录名的 i 节点，返回 0。
104     iput(current->root);
105     current->root = inode;
106     return (0);
107 }
108
109 // 修改文件属性系统调用。
110 // 参数 filename 是文件名，mode 是新的文件属性。
111 // 若操作成功则返回 0，否则返回出错码。
112 int sys_chmod(const char * filename,int mode)
113 {
114     struct m_inode * inode;
115
116     // 该调用为指定文件设置新的访问属性 mode。文件的访问属性在文件名对应的 i 节点中，因此
117     // 我们首先取文件名对应的 i 节点。如果 i 节点不存在，则返回出错码（文件或目录不存在）。
118     // 如果当前进程的有效用户 id 与文件 i 节点的用户 id 不同，并且也不是超级用户，则放回该
119     // 文件 i 节点，返回出错码（没有访问权限）。
120     if (!(inode=namei(filename)))
121         return -ENOENT;
122     if ((current->euid != inode->i_uid) && !suser()) {
123         iput(inode);
124         return -EACCES;
125     }
126 // 否则就重新设置该 i 节点的文件属性，并置该 i 节点已修改标志。放回该 i 节点，返回 0。
127     inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
128     inode->i_dirt = 1;
129     iput(inode);
130     return 0;
131 }
132
133 // 修改文件宿主系统调用。
134 // 参数 filename 是文件名，uid 是用户标识符(用户 ID)，gid 是组 ID。
135 // 若操作成功则返回 0，否则返回出错码。
136 int sys_chown(const char * filename,int uid,int gid)
137 {
138     struct m_inode * inode;
139
140     // 该调用用于设置文件 i 节点中的用户和组 ID，因此首先要取得给定文件名的 i 节点。如果文

```

```

// 件名的 i 节点不存在，则返回出错码（文件或目录不存在）。如果当前进程不是超级用户，
// 则放回该 i 节点，并返回出错码（没有访问权限）。
125     if (!(inode=namei(filename)))
126         return -ENOENT;
127     if (!suser()) {
128         iput(inode);
129         return -EACCES;
130     }
// 否则我们就用参数提供的值来设置文件 i 节点的用户 ID 和组 ID，并置 i 节点已经修改标志，
// 放回该 i 节点，返回 0。
131     inode->i_uid=uid;
132     inode->i_gid=gid;
133     inode->i_dirt=1;
134     iput(inode);
135     return 0;
136 }
137
///// 打开（或创建）文件系统调用。
// 参数 filename 是文件名，flag 是打开文件标志，它可取值：O_RDONLY（只读）、O_WRONLY
// （只写）或 O_RDWR（读写），以及 O_CREAT（创建）、O_EXCL（被创建文件必须不存在）、
// O_APPEND（在文件尾添加数据）等其他一些标志的组合。如果本调用创建了一个新文件，则
// mode 就用于指定文件的许可属性。这些属性有 S_IRWXU（文件宿主具有读、写和执行权限）、
// S_IRUSR（用户具有读文件权限）、S_IRWXG（组成员具有读、写和执行权限）等等。对于新
// 创建的文件，这些属性只应用于将来对文件的访问，创建了只读文件的打开调用也将返回一
// 个可读写的文件句柄。如果调用操作成功，则返回文件句柄（文件描述符），否则返回出错码。
// 参见 sys/stat.h、fcntl.h。
138 int sys_open(const char * filename,int flag,int mode)
139 {
140     struct m_inode * inode;
141     struct file * f;
142     int i,fd;
143
// 首先对参数进行处理。将用户设置的文件模式和进程模式屏蔽码相与，产生许可的文件模式。
// 为了为打开文件建立一个文件句柄，需要搜索进程结构中文件结构指针数组，以查找一个空
// 闲项。空闲项的索引号 fd 即是句柄值。若已经没有空闲项，则返回出错码（参数无效）。
144     mode &= 0777 & ~current->umask;
145     for(fd=0 ; fd<NR_OPEN ; fd++)
146         if (!current->filp[fd]) // 找到空闲项。
147             break;
148     if (fd>=NR_OPEN)
149         return -EINVAL;
// 然后我们设置当前进程的执行时关闭文件句柄（close_on_exec）位图，复位对应的比特位。
// close_on_exec 是一个进程所有文件句柄的位图标志。每个比特位代表一个打开着的文件描
// 述符，用于确定在调用系统调用 execve()时需要关闭的文件句柄。当程序使用 fork()函数
// 创建了一个子进程时，通常会在该子进程中调用 execve()函数加载执行另一个新程序。此时
// 子进程中开始执行新程序。若一个文件句柄在 close_on_exec 中的对应比特位被置位，那么
// 在执行 execve()时该对应文件句柄将被关闭，否则该文件句柄将始终处于打开状态。当打开
// 一个文件时，默认情况下文件句柄在子进程中也处于打开状态。因此这里要复位对应比特位。
150     current->close_on_exec &= ~(1<<fd);

// 然后为打开文件在文件表中寻找一个空闲结构项。我们令 f 指向文件表数组开始处。搜索空
// 闲文件结构项（引用计数为 0 的项），若已经没有空闲文件表结构项，则返回出错码。
// 另外，第 151 行上的指针赋值 "0+file_table" 等同于 "file_table" 和 "&file_table[0]"。

```

```

// 不过这样写可能更能明了一些。
151     f=0+file_table;
152     for (i=0 ; i<NR_FILE ; i++,f++)
153         if (!f->f_count) break;
154     if (i>=NR_FILE)
155         return -EINVAL;
// 此时我们让进程对应文件句柄 fd 的文件结构指针指向搜索到的文件结构，并令文件引用计数
// 递增 1。然后调用函数 open_namei() 执行打开操作，若返回值小于 0，则说明出错，于是释放
// 刚申请到的文件结构，返回出错码 i。若文件打开操作成功，则 inode 是已打开文件的 i 节点
// 指针。
156     (current->filp[fd]=f)->f_count++;
157     if ((i=open_namei(filename,flag,mode,&inode))<0) {
158         current->filp[fd]=NULL;
159         f->f_count=0;
160         return i;
161     }
// 根据已打开文件的 i 节点的属性字段，我们可以知道文件的具体类型。对于不同类型的文件，
// 我们需要作一些特别的处理。如果打开的是字符设备文件，那么，对于主设备号是 4 的字符文
// 件(例如/dev/tty0)，如果当前进程是进程组首领并且当前进程的 tty 字段小于 0（没有终端），
// 则设置当前进程的 tty 号为该 i 节点的子设备号，并设置当前进程 tty 对应的 tty 表项的父进
// 程组号等于当前进程的进程组号。表示为该进程组（会话期）分配控制终端。对于主设备号是
// 5 的字符文件 (/dev/tty)，若当前进程没有 tty，则说明出错，于是放回 i 节点和申请到的
// 文件结构，返回出错码（无许可）。
// （注：这段代码存在问题）。
162 /* ttys are somewhat special (ttyxx major==4, tty major==5) */
/* ttys 有些特殊（ttyxx 的主设备号==4，tty 的主设备号==5）*/
163     if (S_ISCHR(inode->i_mode))
164         if (MAJOR(inode->i_zone[0])==4) {
165             if (current->leader && current->tty<0) {
166                 current->tty = MINOR(inode->i_zone[0]);
167                 tty_table[current->tty].pgrp = current->pgrp;
168             }
169         } else if (MAJOR(inode->i_zone[0])==5)
170             if (current->tty<0) {
171                 iput(inode);
172                 current->filp[fd]=NULL;
173                 f->f_count=0;
174                 return -EPERM;
175             }
176 /* Likewise with block-devices: check for floppy change */
/* 同样对于块设备文件：需要检查盘片是否被更换 */
// 如果打开的是块设备文件，则检查盘片是否更换过。若更换过则需要让高速缓冲区中该设备
// 的所有缓冲块失效。
177     if (S_ISBLK(inode->i_mode))
178         check_disk_change(inode->i_zone[0]);
// 现在我们初始化打开文件的文件结构。设置文件结构属性和标志，置句柄引用计数为 1，并
// 设置 i 节点字段为打开文件的 i 节点，初始化文件读写指针为 0。最后返回文件句柄号。
179     f->f_mode = inode->i_mode;
180     f->f_flags = flag;
181     f->f_count = 1;
182     f->f_inode = inode;
183     f->f_pos = 0;
184     return (fd);

```

```

185 }
186
187 // 创建文件系统调用。
188 // 参数 pathname 是路径名，mode 与上面的 sys_open() 函数相同。
189 // 成功则返回文件句柄，否则返回出错码。
190 int sys_creat(const char * pathname, int mode)
191 {
192     return sys_open(pathname, O_CREAT | O_TRUNC, mode);
193 }
194
195 // 关闭文件系统调用。
196 // 参数 fd 是文件句柄。
197 // 成功则返回 0，否则返回出错码。
198 int sys_close(unsigned int fd)
199 {
200     struct file * filp;
201
202     // 首先检查参数有效性。若给出的文件句柄值大于程序同时能打开的文件数 NR_OPEN，则返回
203     // 出错码（参数无效）。然后复位进程的运行时关闭文件句柄位图对应位。若该文件句柄对应
204     // 的文件结构指针是 NULL，则返回出错码。
205     if (fd >= NR_OPEN)
206         return -EINVAL;
207     current->close_on_exec &= ~(1<<fd);
208     if (!(filp = current->filp[fd]))
209         return -EINVAL;
210
211     // 现在置该文件句柄的文件结构指针为 NULL。若在关闭文件之前，对应文件结构中的句柄引用
212     // 计数已经为 0，则说明内核出错，停机。否则将对应文件结构的引用计数减 1。此时如果它还
213     // 不为 0，则说明有其他进程正在使用该文件，于是返回 0（成功）。如果引用计数已等于 0，
214     // 说明该文件已经没有进程引用，该文件结构已变为空闲。则释放该文件 i 节点，返回 0。
215     current->filp[fd] = NULL;
216     if (filp->f_count == 0)
217         panic("Close: file count is 0");
218     if (--filp->f_count)
219         return (0);
220     iput(filp->f_inode);
221     return (0);
222 }
223

```

12.16 exec.c 程序

12.16.1 功能描述

本源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()` 函数簇的内核实现函数。其它 5 个相关 `exec` 函数一般在库函数中实现，并最终都要调用这个系统调用。当一个程序使用 `fork()` 函数创建了一个子进程时，通常会在该子进程中调用 `exec()` 簇函数之一以加载执行另一个新程序。此时子进程的代码、数据段（包括堆、栈内容）将完全被新程序的替换掉，并在子进程中开始执行新程序。`execve()` 函数的主要功能为：

- 执行对命令行参数和环境参数空间页面的初始化操作 -- 设置初始空间起始指针；初始化空间页面指针数组为(NULL)；根据执行文件名取执行对象的 i 节点；计算参数个数和环境变量个数；检查文件类型，执行权限；
- 根据执行文件开始部分的执行头数据结构，对其中信息进行处理 -- 根据被执行文件 i 节点读取文件头部信息；若是 Shell 脚本程序（第一行以#!开始），则分析 Shell 程序名及其参数，并以被执行文件作为参数执行该执行的 Shell 程序；执行根据文件的幻数以及段长度等信息判断是否可执行；
- 对当前调用进程进行运行新文件前初始化操作 -- 指向新执行文件的 i 节点；复位信号处理句柄；根据头结构信息设置局部描述符基址和段长；设置参数和环境参数页面指针；修改进程各执行字段内容；
- 替换堆栈上原调用 `execve()` 程序的返回地址为新执行程序运行地址，运行新加载的程序。

在 `execve()` 执行过程中，系统会清掉 `fork()` 复制的原程序的页目录和页表项，并释放对应页面。系统仅为新加载的程序代码重新设置进程数据结构中的信息，申请和映射了命令行参数和环境参数块所占的内存页面，以及设置了执行代码执行点。此时内核并不从执行文件所在块设备上加载程序的代码和数据。当该过程返回时即开始执行新的程序，但一开始执行肯定会引起缺页异常中断发生。因为代码和数据还未被从块设备上读入内存。此时缺页异常处理过程会根据引起异常的线性地址在主内存区为新程序申请内存页面（内存帧），并从块设备上读入引起异常的指定页面。同时还为该线性地址设置对应的页目录项和页表项。这种加载执行文件的方法称为需求加载（Load on demand），参见内存管理一章中的说明。

另外，由于新程序是在子进程中执行，所以该子进程就是新程序的进程。新程序的进程 ID 就是该子进程的进程 ID。同样，该子进程的属性也就成为了新程序进程的属性。而对于已打开文件的处理则与每个文件描述符的执行时关闭（close on exec）标志有关。参见对文件 `linux/fs/fcntl.c` 的说明。进程中每个打开的文件描述符都有一个执行时关闭标志。在进程控制结构中是使用一个无符号长整数 `close_on_exec` 来表示的。它的每个比特位表示对应每个文件描述符的该标志。若一个文件描述符在 `close_on_exec` 中的对应比特位被设置，那么在执行 `execve()` 时该描述符将被关闭，否则该描述符将始终处于打开状态。除非我们使用了文件控制函数 `fcntl()` 特别地设置了该标志，否则内核默认操作在 `execve` 执行后仍然保持描述符的打开状态。

关于命令行参数和环境参数的含义解释如下。当用户在命令提示符下键入一个命令时，所指定执行的程序会从该命令行上接受键入的命令行参数。例如当用户键入以下文件名列表命令时：

```
ls -l /home/john/
```

shell 进程会创建一个新进程并在其中执行 `/bin/ls` 命令。在加载 `/bin/ls` 执行文件时命令行上的三个参数 `ls`、`-l` 和 `/home/john/` 将被新进程继承下来。在支持 C 的环境中，当调用程序的主函数 `main()` 时它会带有两个参数。

```
int main(int argc, char *argv[])
```

第一个是执行程序时命令行上参数的个数值，通常记为 `argc`（argument count），第二个是指向包含字符串参数的指针数组（`argv` -- argument vector）。每个字符串代表一个参数，并且 `argv` 数组的结尾总是以空指针来结束。通常，`argv[0]` 是被执行的程序名，因此 `argc` 的值至少是 1。对于上面的例子，此时 `argc=3`，`argv[0]`、`argv[1]` 和 `argv[2]` 分别是 `'ls'`、`'-l'` 和 `'/home/john/'`。而 `argv[3] = NULL`。见图 12-30 所示。

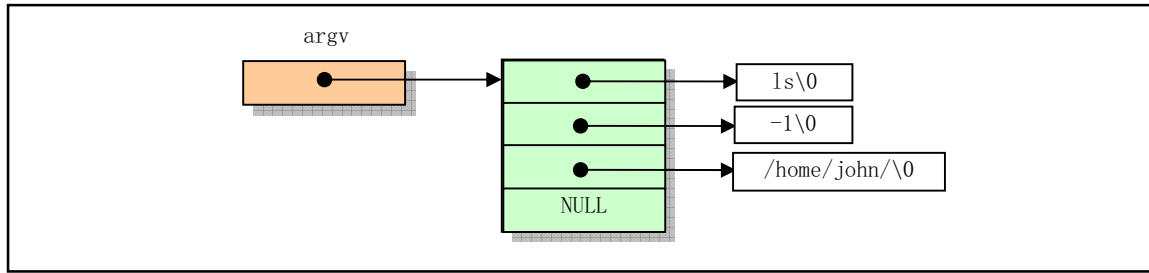


图 12-30 命令行参数指针数组 argv[]

main()还有第三个可选参数，该参数中包含环境变量（environment variable）参数，用于定制执行程序的环境设置并为其提供环境设置参数值。它也是一个指向包含字符串参数的指针数组，并以 NULL 结束，只是这些字符串是环境变量值。当程序需要明确用到环境变量时，main()的声明为：

```
int main(int argc, char *argv[], char *envp[])
```

环境字符串的形式为：

```
VAR_NAME=somevalue
```

其中 VAR_NAME 表示一个环境变量的名称，而等号后面的串代表给这个环境变量所赋的值。在命令行提示符下键入 shell 内部命令 set 可以显示出当前环境中设置的环境参数列表。在程序开始执行前，命令行参数和环境字符串被放置在用户堆栈顶端的地方，见下面说明。

execve() 函数有大量对命令行参数和环境空间的处理操作，参数和环境空间共可有 MAX_ARG_PAGES 个页面，总长度可达 128kB 字节。在该空间中存放数据的方式类似于堆栈操作，即从假设的 128kB 空间末端处逆向开始存放参数或环境变量字符串的。在初始时，程序定义了一个指向该空间末端(128kB-4 字节)处空间内偏移值 p，该偏移值随着存放数据的增多而后退，由图 12-31 中可以看出，p 明确地指出了当前参数环境空间中还剩余多少可用空间。copy_string() 函数用于从用户内存空间拷贝命令行参数和环境字符串到内核空闲页面中。在分析函数 copy_string() 时，可参照此图。

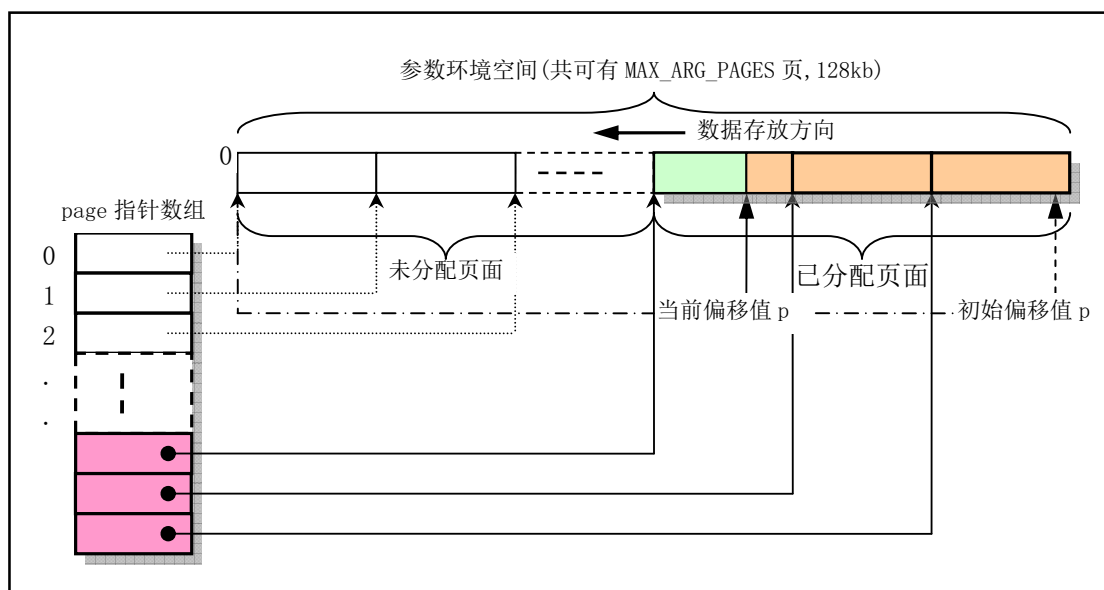
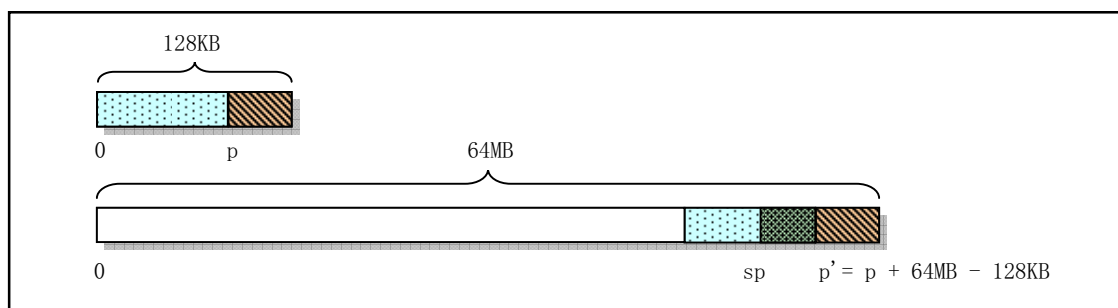


图 12-31 参数和环境变量字符串空间

在执行完 `copy_string()` 之后，再通过执行第 333 行语句，`p` 将被调整为从进程逻辑地址空间开始处算起的参数和环境变量起始处指针，见图 12-32 中所示的 `p'`。方法是把一个进程占用的最大逻辑空间长度 64MB 减去参数和环境变量占用的长度 (128KB - `p`)。 `p'` 的左边部分还将使用 `create_tables()` 函数来存放参数和环境变量的一个指针表，并且 `p'` 将再次向左调整为指向指针表的起始位置处。再把所得指针进行页面对齐，最终得到初始堆栈指针 `sp`。

图 12-32 `p` 转换成进程初始堆栈指针的方法

`create_tables()` 函数用于根据给定的当前堆栈指针值 `p` 以及参数变量个数 `argc` 和环境变量个数 `envc`，在新的程序堆栈中创建环境和参数变量指针表，并返回此时的堆栈指针值，再把该指针进行页面对齐处理，最终得到初始堆栈指针 `sp`。创建完毕后堆栈指针表的形式见下图 12-33 所示。

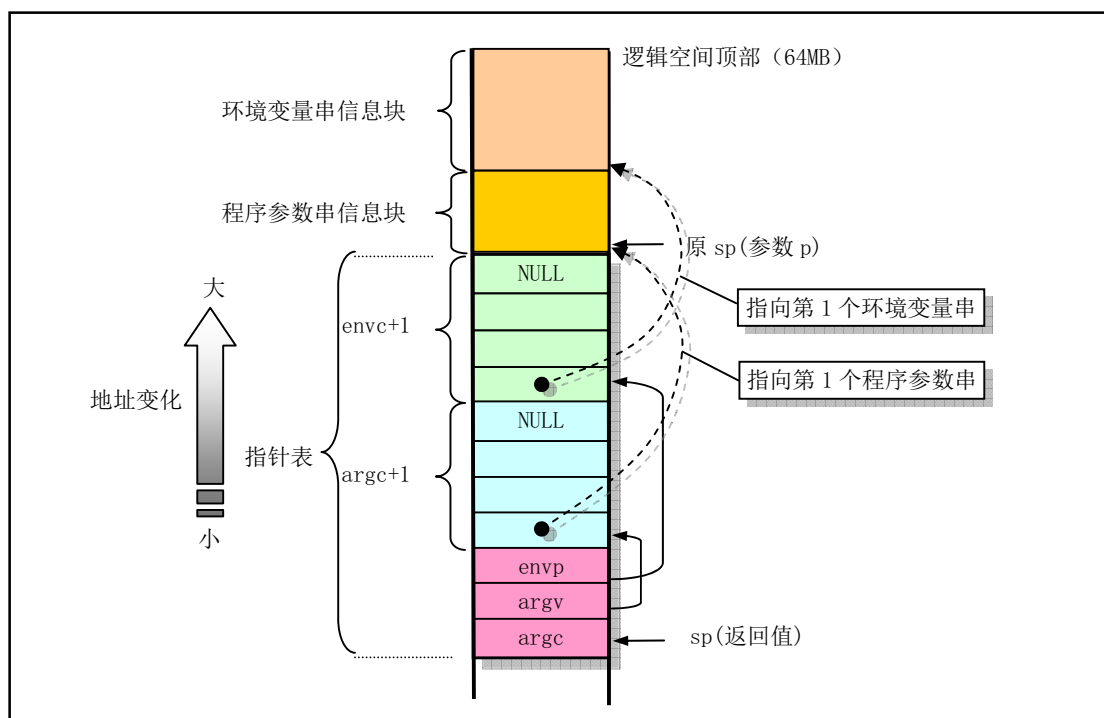


图 12-33 新程序堆栈中指针表示意图

函数 `do_execve()` 最后返回时（第 344、345 行）会把原调用系统中断程序在堆栈上的代码指针 `eip` 替换为指向新执行程序的入口点，并将栈指针替换为新执行文件的栈指针 `esp`。此后这次系统调用的返回指令最终会弹出这些栈中数据，并使得 CPU 去执行新执行文件。这个过程的示意图见图 12-34 所示。图中左半部分是进程逻辑 64MB 的空间还包含原执行程序时的情况；右半部分是释放了原执行程序代码和数据并且更新的堆栈和代码指针时的情况。其中阴影（彩色）部分中包含代码或数据信息。进程任务结构中的 `start_code` 是 CPU 线性空间中的地址，其余几个变量值均是进程逻辑空间中的地址。

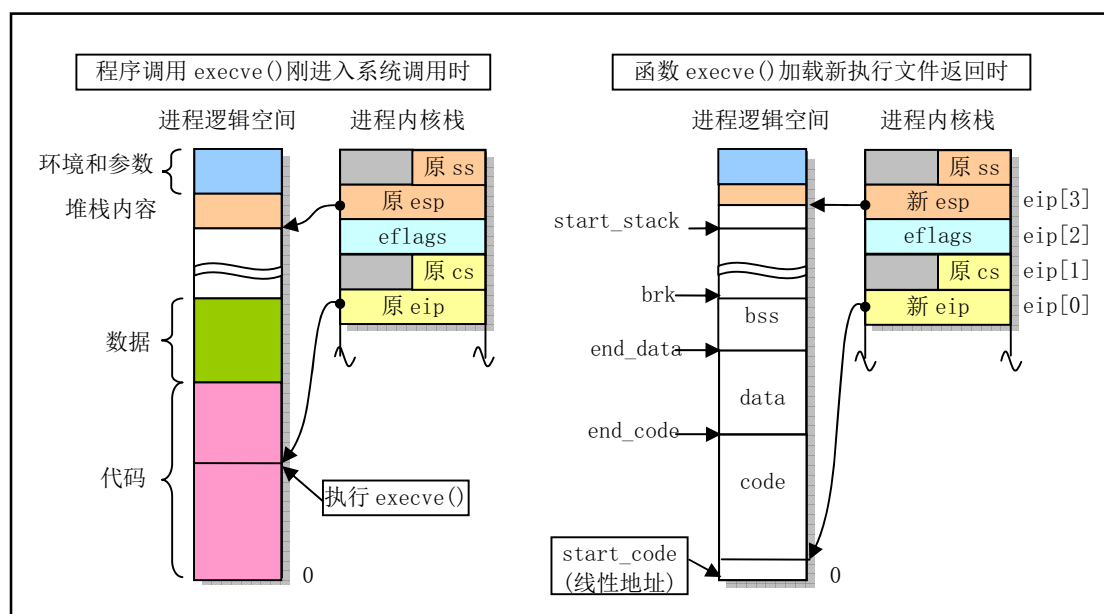


图 12-34 加载执行文件过程中栈中 esp 和 eip 的变化

12.16.2 代码注释

程序 12-15 linux/fs/exec.c

```

1  /*
2  *  linux/fs/exec.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  #-checking implemented by tytso.
9  */
10 /*
11 *  #-开始的脚本程序的检测代码部分是由 tytso 实现的。
12 */
13
14 /* Demand-loading implemented 01.12.91 - no need to read anything but
15 * the header into memory. The inode of the executable is put into
16 * "current->executable", and page faults do the actual loading. Clean.
17 *
18 * Once more I can proudly say that linux stood up to being changed: it
19 * was less than 2 hours work to get demand-loading completely implemented.
20 */
21
22 /* 需求时加载实现于 1991. 12. 1 - 只需将执行文件头部读进内存而无须将整个
23 * 执行文件都加载进内存。执行文件的 i 节点被放在当前进程的可执行字段中
24 * "current->executable", 页异常会进行执行文件的实际加载操作。这很完美。
25 *
26 * 我可以再一次自豪地说, linux 经得起修改: 只用了不到 2 小时的工作时间就
27 * 完全实现了需求加载处理。
28 */
29
30 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。
31 #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。
32 #include <sys/stat.h>       // 文件状态头文件。含有文件状态结构 stat {} 和符号常量等。
33 #include <a.out.h>          // a.out 头文件。定义了 a.out 执行文件格式和一些宏。
34
35 #include <linux/fs.h>        // 文件系统头文件。定义文件表结构 (file、m_inode) 等。
36 #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。
37 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
38 #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。
39 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
40
41 extern int sys_exit(int exit_code);    // 程序退出系统调用。
42 extern int sys_close(int fd);          // 文件关闭系统调用。
43
44 /*
45 *  MAX_ARG_PAGES defines the number of pages allocated for arguments
46 *  and envelope for the new program. 32 should suffice, this gives

```

```

37  * a maximum env+arg of 128kB !
38  */
/*
 * MAX_ARG_PAGES 定义了为新程序分配的给参数和环境变量使用的最大内存页数。
 * 32 页内存应该足够了，这使得环境和参数 (env+arg) 空间的总和达到 128kB!
 */
39 #define MAX_ARG_PAGES 32
40
41 /*
42  * create_tables() parses the env- and arg-strings in new user
43  * memory and creates the pointer tables from them, and puts their
44  * addresses on the "stack", returning the new stack pointer value.
45  */
/*
 * create_tables() 函数在新任务内存中解析环境变量和参数字符串，由此
 * 创建指针表，并将它们的地址放到“栈”上，然后返回新栈的指针值。
 */
///// 在新任务栈中创建参数和环境变量指针表。
// 参数: p - 数据段中参数和环境信息偏移指针; argc - 参数个数; envc - 环境变量个数。
// 返回: 栈指针值。
46 static unsigned long * create_tables(char * p, int argc, int envc)
47 {
48     unsigned long *argv, *envp;
49     unsigned long * sp;
50
51     // 栈指针是以 4 字节 (1 节) 为边界进行寻址的，因此这里需让 sp 为 4 的整数倍值。此时 sp
52     // 位于参数环境表的末端。然后我们先把 sp 向下 (低地址方向) 移动，在栈中空出环境变量
53     // 指针占用的空间，并让环境变量指针 envp 指向该处。多空出的一个位置用于在最后存放一
54     // 个 NULL 值。下面指针加 1，sp 将递增指针宽度字节值 (4 字节)。再把 sp 向下移动，空出
55     // 命令行参数指针占用的空间，并让 argv 指针指向该处。同样，多空处的一个位置用于存放
56     // 一个 NULL 值。此时 sp 指向参数指针块的起始处，我们将环境参数块指针 envp 和命令行参
57     // 数块指针以及命令行参数个数值分别压入栈中。
51     sp = (unsigned long *) (0xffffffffc & (unsigned long) p);
52     sp -= envc+1; // 即 sp = sp - (envc+1);
53     envp = sp;
54     sp -= argc+1;
55     argv = sp;
56     put_fs_long((unsigned long)envp, --sp);
57     put_fs_long((unsigned long)argv, --sp);
58     put_fs_long((unsigned long)argc, --sp);
59     // 再将命令行各参数指针和环境变量各指针分别放入前面空出来的相应地方，最后分别放置一
60     // 个 NULL 指针。
59     while (argc-->0) {
60         put_fs_long((unsigned long) p, argv++);
61         while (get_fs_byte(p++)) /* nothing */; // p 指针指向下一个参数串。
62     }
63     put_fs_long(0, argv);
64     while (envc-->0) {
65         put_fs_long((unsigned long) p, envp++);
66         while (get_fs_byte(p++)) /* nothing */; // p 指针指向下一个环境串。
67     }
68     put_fs_long(0, envp);
69     return sp; // 返回构造的当前新栈指针。

```

```

70 }
71
72 /*
73  * count() counts the number of arguments/envelopes
74  */
75 /*
76  * count() 函数计算命令行参数/环境变量的个数。
77  */
78 // 计算参数个数。
79 // 参数: argv - 参数指针数组, 最后一个指针项是 NULL。
80 // 统计参数指针数组中指针的个数。关于函数参数传递指针的指针的作用, 请参见程序
81 // kernel/sched.c 中第 151 行前的注释。
82 // 返回: 参数个数。
83 static int count(char ** argv)
84 {
85     int i=0;
86     char ** tmp;
87
88     if (tmp = argv)
89         while (get_fs_long((unsigned long *) (tmp++)))
90             i++;
91
92     return i;
93 }
94
95 /*
96  * 'copy_string()' copies argument/envelope strings from user
97  * memory to free pages in kernel mem. These are in a format ready
98  * to be put directly into the top of new user memory.
99  *
100  * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
101  * whether the string and the string array are from user or kernel segments:
102  *
103  * from_kmem    argv *    argv **
104  * 0            user space user space
105  * 1            kernel space user space
106  * 2            kernel space kernel space
107  *
108  * We do this by playing games with the fs segment register. Since it
109  * it is expensive to load a segment register, we try to avoid calling
110  * set_fs() unless we absolutely have to.
111  */
112 /*
113  * 'copy_string()' 函数从用户内存空间拷贝参数/环境字符串到内核空闲页面中。
114  * 这些已具有直接放到新用户内存中的格式。
115  *
116  * 由 TYT(Tytso) 于 1991. 11. 24 日修改, 增加了 from_kmem 参数, 该参数指明了
117  * 字符串或字符串数组是来自用户段还是内核段。
118  *
119  * from_kmem    指针 argv *    字符串 argv **
120  * 0            用户空间      用户空间
121  * 1            内核空间      用户空间
122  * 2            内核空间      内核空间

```

```

*
* 我们是通过巧妙处理 fs 段寄存器来操作的。由于加载一个段寄存器代价太高，
* 所以我们尽量避免调用 set_fs()，除非实在必要。
*/
//// 复制指定个数的参数字符串到参数和环境空间中。
// 参数：argc - 欲添加的参数个数；argv - 参数指针数组；page - 参数和环境空间页面指针
// 数组。p - 参数空间中偏移指针，始终指向已复制串的头部；from_kmem - 字符串来源标志。
// 在 do_execve() 函数中，p 初始化为指向参数表(128kB)空间的最后一个长字处，参数字符串
// 是以堆栈操作方式逆向往其中复制存放的。因此 p 指针会随着复制信息的增加而逐渐减小，
// 并始终指向参数字符串的头部。字符串来源标志 from_kmem 应该是 TYT 为了给 execve() 增添
// 执行脚本文件的功能而新加的参数。当没有运行脚本文件的功能时，所有参数字符串都在用
// 户数据空间中。
// 返回：参数和环境空间当前头部指针。若出错则返回 0。
104 static unsigned long copy_strings(int argc, char ** argv, unsigned long *page,
105                                unsigned long p, int from_kmem)
106 {
107     char *tmp, *pag;
108     int len, offset = 0;
109     unsigned long old_fs, new_fs;
110
111     // 首先取当前段寄存器 ds（指向内核数据段）和 fs 值，分别保存到变量 new_fs 和 old_fs 中。
112     // 如果字符串和字符串数组（指针）来自内核空间，则设置 fs 段寄存器指向内核数据段。
113     if (!p)
114         return 0;
115     new_fs = get_ds();
116     old_fs = get_fs();
117     if (from_kmem==2) // 若串及其指针在内核空间则设置 fs 指向内核空间。
118         set_fs(new_fs);
119     // 然后循环处理各个参数，从最后一个参数逆向开始复制，复制到指定偏移地址处。在循环中，
120     while (argc-- > 0) {
121         // 首先取需要复制的当前字符串指针。如果字符串在用户空间而字符串数组（字符串指针）在
122         // 内核空间，则设置 fs 段寄存器指向内核数据段（ds）。并在内核数据空间中取了字符串指针
123         // tmp 之后就立刻恢复 fs 段寄存器原值（fs 再指回用户空间）。否则不用修改 fs 值而直接从
124         // 用户空间取字符串指针到 tmp。
125         if (from_kmem == 1) // 若串指针在内核空间，则 fs 指向内核空间。
126             set_fs(new_fs);
127         if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
128             panic("argc is wrong");
129         if (from_kmem == 1) // 若串指针在内核空间，则 fs 指回用户空间。
130             set_fs(old_fs);
131         // 然后从用户空间取该字符串，并计算该参数字符串长度 len。此后 tmp 指向该字符串末端。
132         // 如果该字符串长度超过此时参数和环境空间中还剩余的空闲长度，则空间不够了。于是恢复
133         // fs 段寄存器值（如果被改变的话）并返回 0。不过因为参数和环境空间留有 128KB，所以通
134         // 常不可能发生这种情况。
135         len=0;
136         do {
137             len++;
138         } while (get_fs_byte(tmp++));
139         if (p-len < 0) {
140             // this shouldn't happen - 128kB
141             set_fs(old_fs);
142             // 不会发生-因为有 128kB 的空间
143             return 0;
144         }
145     }
146     // 接着我们逆向逐个字符地把字符串复制到参数和环境空间末端处。在循环复制字符串的字符

```



```

// 过程中, 我们首先要判断参数和环境空间中相应位置处是否已经有内存页面。如果还没有就
// 先为其申请 1 页内存页面。偏移量 offset 被用作在一个页面中的当前指针偏移值。 因为
// 刚开始执行本函数时, 偏移变量 offset 被初始化为 0, 所以(offset-1 < 0)肯定成立而使得
// offset 重新被设置为当前 p 指针在页面范围内的偏移值。
132         while (len) {
133             --p; --tmp; --len;
134             if (--offset < 0) {
135                 offset = p % PAGE\_SIZE;
// 如果字符串和字符串数组都在内核空间中, 那么为了从内核数据空间复制字符串内容, 下面
// 143 行上会把 fs 设置为指向内核数据段。而这里 137 行语句则是用来恢复 fs 段寄存器原值。
136                 if (from_kmem==2) // 若字符串在内核空间。
137                     set\_fs(old_fs);
// 如果当前偏移值 p 所在的串空间页面指针数组项 page[p/PAGE_SIZE] ==0, 表示此时 p 指针
// 所处的空间内存页面还不存在, 则需申请一空闲内存页, 并将该页面指针填入指针数组, 同
// 时也使页面指针 pag 指向该新页面。若申请不到空闲页面则返回 0。
138                 if (!(pag = (char *) page[p/PAGE\_SIZE]) &&
139                     !(pag = (char *) page[p/PAGE\_SIZE] =
140                         (unsigned long *) get\_free\_page()))
141                     return 0;
// 如果字符串在内核空间, 则设置 fs 段寄存器指向内核数据段 (ds)。
142                 if (from_kmem==2)
143                     set\_fs(new_fs);
144             }
145         }
// 然后从 fs 段中复制字符串的 1 字节到参数和环境空间内存页面 pag 的 offset 处。
146         *(pag + offset) = get\_fs\_byte(tmp);
147     }
148 }
// 如果字符串和字符串数组在内核空间, 则恢复 fs 段寄存器原值。最后, 返回参数和环境空
// 间中已复制参数的头部偏移值。
149     if (from_kmem==2)
150         set\_fs(old_fs);
151     return p;
152 }
153
//// 修改任务的局部描述符表内容。
// 修改局部描述符表 LDT 中描述符的段基址和段限长, 并将参数和环境空间页面放置在数据段
// 末端。
// 参数: text_size - 执行文件头部中 a_text 字段给出的代码段长度值;
// page - 参数和环境空间页面指针数组。
// 返回: 数据段限长值 (64MB)。
154 static unsigned long change\_ldt(unsigned long text_size,unsigned long * page)
155 {
156     unsigned long code_limit,data_limit,code_base,data_base;
157     int i;
158
// 首先根据执行文件头部代码长度字段 a_text 值, 计算以页面长度为边界的代码段限长。并
// 设置数据段长度为 64MB。 然后取当前进程局部描述符表代码段描述符中代码段基址, 代码
// 段基址与数据段基址相同。并使用这些新值重新设置局部表中代码段和数据段描述符中的基
// 址和段限长。这里请注意, 由于被加载的新程序的代码和数据段基址与原程序的相同, 因此
// 没有必要再重复去设置它们, 即第 164 和 166 行上两条设置段基址的语句多余, 可省略。
159     code_limit = text_size+PAGE\_SIZE -1;
160     code_limit &= 0xFFFFF000;

```

```

161     data_limit = 0x4000000;
162     code_base = get\_base(current->ldt[1]); // include/linux/sched.h, 第 226 行。
163     data_base = code_base;
164     set\_base(current->ldt[1], code_base); // sched.h, 第 211 行。
165     set\_limit(current->ldt[1], code_limit); // sched.h, 第 212 行。
166     set\_base(current->ldt[2], data_base);
167     set\_limit(current->ldt[2], data_limit);
168     /* make sure fs points to the NEW data segment */
169     /* 要确信 fs 段寄存器已指向新的数据段 */
170     // fs 段寄存器中放入局部表数据段描述符的选择符(0x17)。即默认情况下 fs 都指向任务数据段。
171     \_\_asm\_\_ ("pushl $0x17\n\tpop %%fs");
172     // 然后将参数和环境空间已存放数据的页面（最多有 MAX_ARG_PAGES 页，128kB）放到数据段
173     // 末端。方法是从进程空间末端逆向一页一页地放。函数 put_page() 用于把物理页面映射到进
174     // 程逻辑空间中。在 mm/memory.c, 第 197 行。
175     data_base += data_limit;
176     for (i=MAX_ARG_PAGES-1; i>=0; i--) {
177         data_base -= PAGE\_SIZE;
178         if (page[i]) // 若该页面存在，就放置该页面。
179             put\_page(page[i], data_base);
180     }
181     return data_limit; // 最后返回数据段限长(64MB)。
182 }
183
184 /*
185  * 'do_execve()' executes a new program.
186  */
187 /*
188  * 'do_execve()' 函数执行一个新程序。
189  */
190 //// execve() 系统中中断调用函数。加载并执行子进程（其他程序）。
191 // 该函数是系统中中断调用 (int 0x80) 功能号 __NR_execve 调用的函数。函数的参数是进入系统
192 // 调用处理过程后直到调用本系统调用处理过程 (system_call.s 第 200 行) 和调用本函数之前
193 // (system_call.s, 第 203 行) 逐步压入栈中的值。这些值包括：
194 // ① 第 86--88 行入堆的 edx、ecx 和 ebx 寄存器值，分别对应**envp、**argv 和*filename；
195 // ② 第 94 行调用 sys_call_table 中 sys_execve 函数（指针）时压入栈的函数返回地址（tmp）；
196 // ③ 第 202 行在调用本函数 do_execve 前入栈的指向栈中调用系统中中断的程序代码指针 eip。
197 // 参数：
198 // eip - 调用系统中中断的程序代码指针，参见 kernel/system_call.s 程序开始部分的说明；
199 // tmp - 系统中中断中在调用 _sys_execve 时的返回地址，无用；
200 // filename - 被执行程序文件名指针；
201 // argv - 命令行参数指针数组的指针；
202 // envp - 环境变量指针数组的指针。
203 // 返回：如果调用成功，则不返回；否则设置出错号，并返回-1。
204 int do\_execve(unsigned long * eip, long tmp, char * filename,
205               char ** argv, char ** envp)
206 {
207     struct m\_inode * inode; // 内存中 i 节点指针。
208     struct buffer\_head * bh; // 高速缓存块头指针。
209     struct exec ex; // 执行文件头部数据结构变量。
210     unsigned long page[MAX\_ARG\_PAGES]; // 参数和环境串空间页面指针数组。
211     int i, argc, envc;
212     int e_uid, e_gid; // 有效用户 ID 和有效组 ID。
213     int retval; // 返回值。

```

```

192     int sh_bang = 0;                                // 控制是否需要执行脚本程序。
193     unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4;// p 指向参数和环境空间的最后部。
194
// 在正式设置执行文件的运行环境之前，让我们先干些杂事。内核准备了 128KB（32 个页面）
// 空间来存放执行文件的命令行参数和环境字符串。上行把 p 初始设置成位于 128KB 空间的
// 最后 1 个长字处。在初始参数和环境空间的操作过程中，p 将用来指明在 128KB 空间中的当
// 前位置。
// 另外，参数 eip[1]是调用本次系统调用的原用户程序代码段寄存器 CS 值，其中的段选择符
// 当然必须是当前任务的代码段选择符（0x000f）。若不是该值，那么 CS 只能会是内核代码
// 段的选择符 0x0008。但这是绝对不允许的，因为内核代码是常驻内存而不能被替换掉的。
// 因此下面根据 eip[1] 的值确认是否符合正常情况。然后再初始化 128KB 的参数和环境串空
// 间，把所有字节清零，并取出执行文件的 i 节点。再根据函数参数分别计算出命令行参数和
// 环境字符串的个数 argc 和 envc。另外，执行文件必须是常规文件。
195     if ((0xffff & eip[1]) != 0x000f)
196         panic("execve called from supervisor mode");
197     for (i=0 ; i<MAX_ARG_PAGES ; i++)                /* clear page-table */
198         page[i]=0;
199     if (!(inode=namei(filename)))                    /* get executables inode */
200         return -ENOENT;
201     argc = count(argv);                               // 命令行参数个数。
202     envc = count(envp);                               // 环境字符串变量个数。
203
204 restart_interp:
205     if (!S_ISREG(inode->i_mode)) {                    /* must be regular file */
206         retval = -EACCES;
207         goto exec_error2;                            //若不是常规文件则置出错码，跳转到 347 行。
208     }

// 下面检查当前进程是否有权运行指定的执行文件。即根据执行文件 i 节点中的属性，看看本
// 进程是否有权执行它。在把执行文件 i 节点的属性字段值取到 i 中后，我们首先查看属性中
// 是否设置了“设置-用户-ID”（set-user-id）标志和“设置-组-ID”（set-group-id）标
// 志。这两个标志主要是让一般用户能够执行特权用户（如超级用户 root）的程序，例如改变
// 密码的程序 passwd 等。如果 set-user-id 标志置位，则后面执行进程的有效用户 ID（euid）
// 就设置成执行文件的用户 ID，否则设置成当前进程的 euid。如果执行文件 set-group-id 被
// 置位的话，则执行进程的有效组 ID（egid）就设置成执行文件的组 ID。否则设置成当前进程
// 的 egid。这里暂时把这两个判断出来的值保存在变量 e_uid 和 e_gid 中。
209     i = inode->i_mode;                                // 取文件属性字段值。
210     e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
211     e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
// 现在根据进程的 euid 和 egid 和执行文件的访问属性进行比较。如果执行文件属于运行进程
// 的用户，则把文件属性值 i 右移 6 位，此时其最低 3 位是文件宿主的访问权限标志。否则的
// 话如果执行文件与当前进程的用户属于同组，则使属性值最低 3 位是执行文件组用户的访问
// 权限标志。否则此时属性字最低 3 位就是其他用户访问该执行文件的权限。
// 然后我们根据属性字 i 的最低 3 比特值来判断当前进程是否有权运行这个执行文件。如果
// 选出的相应用户没有运行改文件的权力（位 0 是执行权限），并且其他用户也没有任何权限
// 或者当前进程用户不是超级用户，则表明当前进程没有权力运行这个执行文件。于是置不可
// 执行出错码，并跳转到 exec_error2 处去作退出处理。
212     if (current->euid == inode->i_uid)
213         i >>= 6;
214     else if (current->egid == inode->i_gid)
215         i >>= 3;
216     if (!(i & 1) &&
217         !((inode->i_mode & 0111) && suser())) {

```

```

218         retval = -ENOEXEC;
219         goto exec_error2;
220     }
    // 程序执行到这里，说明当前进程有运行指定执行文件的权限。因此从这里开始我们需要取出
    // 执行文件头部数据并根据其中的信息来分析设置运行环境，或者运行另一个 shell 程序来执
    // 行脚本程序。首先读取执行文件第 1 块数据到高速缓冲块中。并复制缓冲块数据到 ex 中。
    // 如果执行文件开始的两个字节是字符'#!'，则说明执行文件是一个脚本文本文件。如果想运
    // 行脚本文件，我们就需要执行脚本文件的解释程序（例如 shell 程序）。通常脚本文件的第
    // 一行文本为“#!/bin/bash”。它指明了运行脚本文件需要的解释程序。运行方法是从脚本
    // 文件第 1 行（带字符'#!'）中取出其中的解释程序名及后面的参数（若有的话），然后将这
    // 些参数和脚本文件名放进执行文件（此时是解释程序）的命令行参数空间中。在这之前我们
    // 当然需要先把函数指定的原有命令行参数和环境字符串放到 128KB 空间中，而这里建立起来
    // 的命令行参数则放到它们前面位置处（因为是逆向放置）。最后让内核执行脚本文件的解释
    // 程序。下面就是在设置好解释程序的脚本文件名等参数后，取出解释程序的 i 节点并跳转到
    // 204 行去执行解释程序。由于我们需要跳转到执行过的代码 204 行去，因此在下面确认并处
    // 理了脚本文件之后需要设置一个禁止再次执行下面的脚本处理代码标志 sh_bang。在后面的
    // 代码中该标志也用来表示我们已经设置好执行文件的命令行参数，不要重复设置。
221     if (!(bh = bread(inode->i_dev, inode->i_zone[0]))) {
222         retval = -EACCES;
223         goto exec_error2;
224     }
225     ex = *((struct exec *) bh->b_data); /* read exec-header */ /* 复制执行头 */
226     if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
227         /*
228          * This section does the #! interpretation.
229          * Sorta complicated, but hopefully it will work. -TYT
230          */
231         /*
232          * 这部分处理对'#!'的解释，有些复杂，但希望能工作。-TYT
233          */
234         char buf[1023], *cp, *interp, *i_name, *i_arg;
235         unsigned long old_fs;
236         // 从这里开始，我们从脚本文件中提取解释程序名及其参数，并把解释程序名、解释程序的参数
237         // 和脚本文件名组合放入环境参数块中。首先复制脚本文件头 1 行字符'#!'后面的字符串到 buf
238         // 中，其中含有脚本解释程序名（例如/bin/sh），也可能还包含解释程序的几个参数。然后对
239         // buf 中的内容进行处理。删除开始的空格、制表符。
240         strncpy(buf, bh->b_data+2, 1022);
241         brelse(bh); // 释放缓冲块并放回脚本文件 i 节点。
242         iput(inode);
243         buf[1022] = '\0';
244         if (cp = strchr(buf, '\n')) {
245             *cp = '\0'; // 第 1 个换行符换成 NULL 并去掉空格制表符。
246             for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
247         }
248         if (!cp || *cp == '\0') { // 若该行没有其他内容，则出错。
249             retval = -ENOEXEC; /* No interpreter name found */
250             goto exec_error1; /* 没有找到脚本解释程序名 */
251         }
252         // 此时我们得到了开头是脚本解释程序名的一行内容（字符串）。下面分析该行。首先取第一个
253         // 字符串，它应该是解释程序名，此时 i_name 指向该名称。若解释程序名后还有字符，则它们应
254         // 该是解释程序的参数串，于是令 i_arg 指向该串。

```

```

247     interp = i_name = cp;
248     i_arg = 0;
249     for ( ; *cp && (*cp != ' ') && (*cp != '|t'); cp++) {
250         if (*cp == '/')
251             i_name = cp+1;
252     }
253     if (*cp) {
254         *cp++ = '|0';           // 解释程序名尾添加 NULL 字符。
255         i_arg = cp;           // i_arg 指向解释程序参数。
256     }
257     /*
258      * OK, we've parsed out the interpreter name and
259      * (optional) argument.
260      */
261     /*
262      * OK, 我们已经解析出解释程序的文件名以及(可选的)参数。
263      */
264     // 现在我们要把上面解析出来的解释程序名 i_name 及其参数 i_arg 和脚本文件名作为解释程
265     // 序的参数放进环境和参数块中。不过首先我们需要把函数提供的原来一些参数和环境字符串
266     // 先放进去, 然后再放这里解析出来的。例如对于命令行参数来说, 如果原来的参数是"-arg1
267     // -arg2"、解释程序名是"bash"、其参数是"-iarg1 -iarg2"、脚本文件名(即原来的执行文
268     // 件名)是"example.sh", 那么在放入这里的参数之后, 新的命令行类似于这样:
269     // "bash -iarg1 -iarg2 example.sh -arg1 -arg2"
270     // 这里我们把 sh_bang 标志置上, 然后把函数参数提供的原有参数和环境字符串放入到空间中。
271     // 环境字符串和参数个数分别是 envc 和 argc-1 个。少复制的一个原有参数是原来的执行文件
272     // 名, 即这里的脚本文件名。[[?? 可以看出, 实际上我们不需要去另行处理脚本文件名, 即这
273     // 里完全可以复制 argc 个参数, 包括原来执行文件名(即现在的脚本文件名)。因为它位于同
274     // 一个位置上 ]]。注意! 这里指针 p 随着复制信息增加而逐渐向小地址方向移动, 因此这两个
275     // 复制串函数执行完后, 环境参数串信息块位于程序命令行参数串信息块的上方, 并且 p 指向
276     // 程序的第 1 个参数串。copy_strings() 最后一个参数 (0) 指明参数字符串在用户空间。
277     if (sh_bang++ == 0) {
278         p = copy_strings(envc, envp, page, p, 0);
279         p = copy_strings(--argc, argv+1, page, p, 0);
280     }
281     /*
282      * Splice in (1) the interpreter's name for argv[0]
283      * (2) (optional) argument to interpreter
284      * (3) filename of shell script
285      *
286      * This is done in reverse order, because of how the
287      * user environment and arguments are stored.
288      */
289     /*
290      * 拼接 (1) argv[0] 中放解释程序的名称
291      * (2) (可选的)解释程序的参数
292      * (3) 脚本程序的名称
293      *
294      * 这是以逆序进行处理的, 是由于用户环境和参数的存放方式造成的。
295      */
296     // 接着我们逆向复制脚本文件名、解释程序的参数和解释程序文件名到参数和环境空间中。
297     // 若出错, 则置出错码, 跳转到 exec_error1。另外, 由于本函数参数提供的脚本文件名
298     // filename 在用户空间, 而这里赋予 copy_strings() 的脚本文件名指针在内核空间, 因此
299     // 这个复制字符串函数的最后一个参数(字符串来源标志)需要被设置成 1。若字符串在

```



```

// 内核空间，则 copy_strings() 的最后一个参数要设置成 2，如下面的第 276、279 行。
273     p = copy_strings(1, &filename, page, p, 1);
274     argc++;
275     if (i_arg) {                                // 复制解释程序的多个参数。
276         p = copy_strings(1, &i_arg, page, p, 2);
277         argc++;
278     }
279     p = copy_strings(1, &i_name, page, p, 2);
280     argc++;
281     if (!p) {
282         retval = -ENOMEM;
283         goto exec_error1;
284     }
285     /*
286      * OK, now restart the process with the interpreter's inode.
287      */
288     /*
289      * OK, 现在使用解释程序的 i 节点重启进程。
290      */
// 最后我们取得解释程序的 i 节点指针，然后跳转到 204 行去执行解释程序。为了获得解释程
// 序的 i 节点，我们需要使用 namei() 函数，但是该函数所使用的参数（文件名）是从用户数
// 据空间得到的，即从段寄存器 fs 所指空间中取得。因此在调用 namei() 函数之前我们需要
// 先临时让 fs 指向内核数据空间，以让函数能从内核空间得到解释程序名，并在 namei()
// 返回后恢复 fs 的默认设置。因此这里我们先临时保存原 fs 段寄存器（原指向用户数据段）
// 的值，将其设置成指向内核数据段，然后取解释程序的 i 节点。之后再恢复 fs 的原值。并
// 跳转到 restart_interp (204 行) 处重新处理新的执行文件 — 脚本文件的解释程序。
288     old_fs = get_fs();
289     set_fs(get_ds());
290     if (!(inode=namei(interp))) {                /* get executables inode */
291         set_fs(old_fs);                        /* 取得解释程序的 i 节点 */
292         retval = -ENOENT;
293         goto exec_error1;
294     }
295     set_fs(old_fs);
296     goto restart_interp;                        // 204 行。
297 }

// 此时缓冲块中的执行文件头结构数据已经复制到了 ex 中。于是先释放该缓冲块，并开始对
// ex 中的执行头信息进行判断处理。对于 Linux 0.11 内核来说，它仅支持 ZMAGIC 执行文件格
// 式，并且执行文件代码都从逻辑地址 0 开始执行，因此不支持含有代码或数据重定位信息的
// 执行文件。当然，如果执行文件实在太大会或者执行文件残缺不全，那么我们也不能运行它。
// 因此对于下列情况将不执行程序：如果执行文件不是需求页可执行文件（ZMAGIC）、或者代
// 码和数据重定位部分长度不等于 0、或者（代码段+数据段+堆）长度超过 50MB、或者执行文件
// 长度小于（代码段+数据段+符号表长度+执行头部分）长度的总和。
298     brelse(bh);
299     if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
300         ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
301         inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
302         retval = -ENOEXEC;
303         goto exec_error2;
304     }

// 另外，如果执行文件中代码开始处没有位于 1 个页面（1024 字节）边界处，则也不能执行。
// 因为需求页（Demand paging）技术要求加载执行文件内容时以页面为单位，因此要求执行

```

```

// 文件映像中代码和数据都从页面边界处开始。
305     if (N_TXTOFF(ex) != BLOCK_SIZE) {
306         printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
307         retval = -ENOEXEC;
308         goto exec_error2;
309     }
// 如果 sh_bang 标志没有设置, 则复制指定个数的命令行参数和环境字符串到参数和环境空间
// 中。若 sh_bang 标志已经设置, 则表明是将运行脚本解释程序, 此时环境变量页面已经复制,
// 无须再复制。同样, 若 sh_bang 没有置位而需要复制的话, 那么此时指针 p 随着复制信息增
// 加而逐渐向小地址方向移动, 因此这两个复制串函数执行完后, 环境参数串信息块位于程序
// 参数串信息块的上方, 并且 p 指向程序的第 1 个参数串。事实上, p 是 128KB 参数和环境空
// 间中的偏移值。因此如果 p=0, 则表示环境变量与参数空间页面已经被占满, 容纳不下了。
310     if (!sh_bang) {
311         p = copy_strings(envc, envp, page, p, 0);
312         p = copy_strings(argc, argv, page, p, 0);
313         if (!p) {
314             retval = -ENOMEM;
315             goto exec_error2;
316         }
317     }
318     /* OK, This is the point of no return */
// * OK, 下面开始就没有返回的地方了 */
// 前面我们针对函数参数提供的信息对需要运行执行文件的命令行参数和环境空间进行了设置,
// 但还没有为执行文件做过什么实质性的工作, 即还没有做过为执行文件初始化进程任务结构
// 信息、建立页表等工作。现在我们就来做这些工作。由于执行文件直接使用当前进程的“躯
// 壳”, 即当前进程将被改造成执行文件的进程, 因此我们需要首先释放当前进程占用的某些
// 系统资源, 包括关闭指定的已打开文件、占用的页表和内存页面等。然后根据执行文件头结
// 构信息修改当前进程使用的局部描述符表 LDT 中描述符的内容, 重新设置代码段和数据段描
// 述符的限长, 再利用前面处理得到的 e_uid 和 e_gid 等信息来设置进程任务结构中相关的字
// 段。最后把执行本次系统调用程序的返回地址 eip[] 指向执行文件中代码的起始位置处。这
// 样当本系统调用退出返回后就会去运行新执行文件的代码了。注意, 虽然此时新执行文件代
// 码和数据还没有从文件中加载到内存中, 但其参数和环境块已经在 copy_strings() 中使用
// get_free_page() 分配了物理内存页来保存数据, 并在 change_ldt() 函数中使用 put_page()
// 放到了进程逻辑空间的末端处。另外, 在 create_tables() 中也会由于在用户栈上存放参数
// 和环境指针表而引起缺页异常, 从而内存管理程序也会就此为用户栈空间映射物理内存页。
//
// 这里我们首先放回进程原执行程序的 i 节点, 并且让进程 executable 字段指向新执行文件
// 的 i 节点。然后复位原进程的所有信号处理句柄。[[但对于 SIG_IGN 句柄无须复位, 因此在
// 322 与 323 行之间应该添加 1 条 if 语句: if (current->sa[i].sa_handler != SIG_IGN)]
// 再根据设定的执行时关闭文件句柄 (close_on_exec) 位图标志, 关闭指定的打开文件, 并
// 复位该标志。
319     if (current->executable)
320         iput(current->executable);
321     current->executable = inode;
322     for (i=0 ; i<32 ; i++)
323         current->sigaction[i].sa_handler = NULL;
324     for (i=0 ; i<NR_OPEN ; i++)
325         if ((current->close_on_exec>>i)&1)
326             sys_close(i);
327     current->close_on_exec = 0;
// 然后根据当前进程指定的基址和限长, 释放原来程序的代码段和数据段所对应的内存页表
// 指定的物理内存页面及页表本身。此时新执行文件并没有占用主内存区任何页面, 因此在处
// 理器真正运行新执行文件代码时就会引起缺页异常中断, 此时内存管理程序即会执行缺页处

```

```

// 理而为新执行文件申请内存页面和设置相关页表项，并且把相关执行文件页面读入内存中。
// 如果“上次任务使用了协处理器”指向的是当前进程，则将其置空，并复位使用了协处理器
// 的标志。
328     free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
329     free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
330     if (last_task_used_math == current)
331         last_task_used_math = NULL;
332     current->used_math = 0;

// 然后我们根据新执行文件头结构中的代码长度字段 a_text 的值修改局部表中描述符基址和
// 段限长，并将 128KB 的参数和环境空间页面放置在数据段末端。执行下面语句之后，p 此时
// 更改成以数据段起始处为原点的偏移值，但仍指向参数和环境空间数据开始处，即已转换成
// 为栈指针值。然后调用内部函数 create_tables() 在栈空间中创建环境和参数变量指针表，
// 供程序的 main() 作为参数使用，并返回该栈指针。
333     p += change_ldt(ex.a_text, page) - MAX_ARG_PAGES * PAGE_SIZE;
334     p = (unsigned long) create_tables((char *)p, argc, envc);

// 接着再修改进程各字段值为新执行文件的信息。即令进程任务结构代码尾字段 end_code 等
// 于执行文件的代码段长度 a_text；数据尾字段 end_data 等于执行文件的代码段长度加数
// 据段长度 (a_data + a_text)；并令进程堆结尾字段 brk = a_text + a_data + a_bss。
// brk 用于指明进程当前数据段（包括未初始化数据部分）末端位置。然后设置进程栈开始字
// 段为栈指针所在页面，并重新设置进程的有效用户 id 和有效组 id。
335     current->brk = ex.a_bss +
336         (current->end_data = ex.a_data +
337         (current->end_code = ex.a_text));
338     current->start_stack = p & 0xfffff000;
339     current->euid = e_uid;
340     current->egid = e_gid;

// 如果执行文件代码加数据长度的末端不在页面边界上，则把最后不到 1 页长度的内存空间初
// 始化为零。[[ 实际上由于使用的是 ZMAGIC 格式的-exec 文件，因此代码段和数据段长度均是
// 页面的整数倍长度，因此 343 行不会执行，即 (i&0xfff) = 0。这段代码是 Linux 内核以前
// 版本的残留物：)]
341     i = ex.a_text + ex.a_data;
342     while (i & 0xfff)
343         put_fs_byte(0, (char *) (i++));

// 最后将原调用系统中中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将栈
// 指针替换为新执行文件的栈指针。此后返回指令将弹出这些栈数据并使得 CPU 去执行新执行
// 文件，因此不会返回到原调用系统中中断的程序中去了。
344     eip[0] = ex.a_entry;      /* eip, magic happens :- ) */ /* eip, 魔法起作用了 */
345     eip[3] = p;               /* stack pointer */          /* esp, 堆栈指针 */
346     return 0;
347 exec_error2:
348     iput(inode);              // 放回 i 节点。
349 exec_error1:
350     for (i = 0; i < MAX_ARG_PAGES; i++)
351         free_page(page[i]);    // 释放存放参数和环境串的内存页面。
352     return(retval);            // 返回出错码。
353 }
354

```

12.16.3 其他信息

12.16.3.1 a.out 执行文件格式

Linux 内核 0.11 版仅支持 a.out(Assembly & link editor output)执行文件格式,虽然这种格式目前已经渐渐不用,而使用功能更为齐全的 ELF(Executable and Link Format)格式,但是由于其简单性,作为学习入门的材料正好比较适用。下面全面介绍一下 a.out 格式。

在头文件<a.out.h>中声明了三个数据结构以及一些宏函数。这些数据结构描述了系统上可执行的机器码文件(二进制文件)。

一个执行文件共可有七个部分(七节)组成。按照顺序,这些部分是:

执行头部分(exec header)

执行文件头部分。该部分中含有一些参数,内核使用这些参数将执行文件加载到内存中并执行,而链接程序(ld)使用这些参数将一些二进制目标文件组合成一个可执行文件。这是唯一必要的组成部分。

代码段部分(text segment)

含有程序执行使被加载到内存中的指令代码和相关数据。可以以只读形式进行加载。

数据段部分(data segment)

这部分含有已经初始化过的数据,总是被加载到可读写的内存中。

代码重定位部分(text relocations)

这部分含有供链接程序使用的记录数据。在组合二进制目标文件时用于定位代码段中的指针或地址。

数据重定位部分(data relocations)

与代码重定位部分的作用类似,但是是用于数据段中指针的重定位。

符号表部分(symbol table)

这部分同样含有供链接程序使用的记录数据,用于在二进制目标文件之间对命名的变量和函数(符号)进行交叉引用。

字符串表部分(string table)

该部分含有与符号名相对应的字符串。

每个二进制执行文件均以执行数据结构(exec structure)开始。该数据结构的形式如下:

```
struct exec {
    unsigned long a_midmag;
    unsigned long a_text;
    unsigned long a_data;
    unsigned long a_bss;
    unsigned long a_syms;
    unsigned long a_entry;
    unsigned long a_trsize;
    unsigned long a_drsize;
};
```

各个字段的功能如下:

a_midmag - 该字段含有被 N_GETFLAG()、N_GETMID 和 N_GETMAGIC()访问的子部分,是由链接程序在运行时加载到进程地址空间。宏 N_GETMID()用于返回机器标识符(machine-id),指示出二进制文件将在什么机器上运行。N_GETMAGIC()宏指明魔数,它唯一地确定了二进制执行文件与其他加载的文件之间的区别。字段中必须包含以下值之一:

- OMAGIC - 表示代码和数据段紧随在执行头后面并且是连续存放的。内核将代码和数据段都加载到可读写内存中。

• NMAGIC - 同 OMAGIC 一样，代码和数据段紧随在执行头后面并且是连续存放的。然而内核将代码加载到了只读内存中，并把数据段加载到代码段后下一页可读写内存边界开始。

• ZMAGIC - 内核在必要时从二进制执行文件中加载独立的页面。执行头部、代码段和数据段都被链接程序处理成多个页面大小的块。内核加载的代码页面时只读的，而数据段的页面是可写的。

a_text - 该字段含有代码段的长度值，字节数。

a_data - 该字段含有数据段的长度值，字节数。

a_bss - 含有‘bss 段’的长度，内核用其设置在数据段后初始的 break (brk)。内核在加载程序时，这段可写内存显现出处于数据段后面，并且初始时为全零。

a_syms - 含有符号表部分的字节长度值。

a_entry - 含有内核将执行文件加载到内存中以后，程序执行起始点的内存地址。

a_trsize - 该字段含有代码重定位表的大小，是字节数。

a_drsize - 该字段含有数据重定位表的大小，是字节数。

在 a.out.h 头文件中定义了几个宏，这些宏使用 exec 结构来测试一致性或者定位执行文件中各个部分（节）的位置偏移值。这些宏有：

• N_BADMAG(exec) 如果 a_magic 字段不能被识别，则返回非零值。

• N_TXTOFF(exec) 代码段的起始位置字节偏移值。

• N_DATOFF(exec) 数据段的起始位置字节偏移值。

• N_DRELOFF(exec) 数据重定位表的起始位置字节偏移值。

• N_TRELOFF(exec) 代码重定位表的起始位置字节偏移值。

• N_SYMOFF(exec) 符号表的起始位置字节偏移值。

• N_STROFF(exec) 字符串表的起始位置字节偏移值。

重定位记录具有标准格式，它使用重定位信息(relocation_info)结构来描述：

```
struct relocation_info {
    int          r_address;
    unsigned int r_symbolnum : 24,
                r_pcrel : 1,
                r_length : 2,
                r_extern : 1,
                r_baserel : 1,
                r_jmptable : 1,
                r_relative : 1,
                r_copy : 1;
};
```

该结构中各字段的含义如下：

r_address - 该字段含有需要链接程序处理（编辑）的指针的字节偏移值。代码重定位的偏移值是从代码段开始处计数的，数据重定位的偏移值是从数据段开始处计算的。链接程序会将已经存储在该偏移处的值与使用重定位记录计算出的新值相加。

r_symbolnum - 该字段含有符号表中一个符号结构的序号值（不是字节偏移值）。链接程序在算出符号的绝对地址以后，就将该地址加到正在进行重定位的指针上。（如果 r_extern 比特位是 0，那么情况就不同，见下面。）

r_pcrel - 如果设置了该位，链接程序就认为正在更新一个指针，该指针使用 pc 相关寻址方式，是属于机器码指令部分。当运行程序使用这个被重定位的指针时，该指针的地址被隐式地加到该指针上。

r_length - 该字段含有指针长度的 2 的次方值：0 表示 1 字节长，1 表示 2 字节长，2 表示 4 字节长。

r_extern - 如果被置位，表示该重定位需要一个外部引用；此时链接程序必须使用一个符号地址来更

新相应指针。当该位是 0 时，则重定位是“局部”的；链接程序更新指针以反映各个段加载地址中的变化，而不是反映一个符号值的变化（除非同时设置了 `r_baserel`，见下面）。在这种情况下，`r_symbolnum` 字段的内容是一个 `n_type` 值（见下面）；这类字段告诉链接程序被重定位的指针指向那个段。

`r_baserel` - 如果设置了该位，则 `r_symbolnum` 字段指定的符号将被重定位成全局偏移表(Global Offset Table)中的一个偏移值。在运行时刻，全局偏移表该偏移处被设置为符号的地址。

`r_jmptable` - 如果被置位，则 `r_symbolnum` 字段指定的符号将被重定位成过程链接表(Procedure Linkage Table)中的一个偏移值。

`r_relative` - 如果被置位，则说明此重定位与该目标文件将成为其组成部分的映像文件在运行时被加载的地址相关。这类重定位仅在共享目标文件中出现。

`r_copy` - 如果被置位，该重定位记录指定了一个符号，该符号的内容将被复制到 `r_address` 指定的地方。该复制操作是通过共享目标模块中一个合适的数据项中的运行时刻链接程序完成的。

符号将名称映射为地址（或者更通俗地讲是字符串映射到值）。由于链接程序对地址的调整，一个符号的名称必须用来表示其地址，直到已被赋予一个绝对地址值。符号是由符号表中固定长度的记录以及字符串表中的可变长度名称组成。符号表是 `nlist` 结构的一个数组，如下所示：

```
struct nlist {
    union {
        char    *n_name;
        long    n_strx;
    } n_un;
    unsigned char n_type;
    char          n_other;
    short         n_desc;
    unsigned long  n_value;
};
```

其中各字段的含义为：

`n_un.n_strx` - 含有本符号的名称在字符串表中的字节偏移值。当程序使用 `nlist()` 函数访问一个符号表时，该字段被替换为 `n_un.n_name` 字段，这是内存中字符串的指针。

`n_type` - 用于链接程序确定如何更新符号的值。使用位屏蔽(bitmasks)可以将 `n_type` 字段分割成三个子字段，对于 `N_EXT` 类型位置位的符号，链接程序将它们看作是“外部的”符号，并且允许其他二进制目标文件对它们的引用。`N_TYPE` 屏蔽码用于链接程序感兴趣的比特位：

- ♦ `N_UNDF` - 一个未定义的符号。链接程序必须在其他二进制目标文件中定位一个具有相同名称的外部符号，以确定该符号的绝对数据值。特殊情况下，如果 `n_type` 字段是非零值，并且没有二进制文件定义了这个符号，则链接程序在 `BSS` 段中将该符号解析为一个地址，保留长度等于 `n_value` 的字节。如果符号在多于一个二进制目标文件中都没有定义并且这些二进制目标文件对其长度值不一致，则链接程序将选择所有二进制目标文件中最大的长度。
- ♦ `N_ABS` - 一个绝对符号。链接程序不会更新一个绝对符号。
- ♦ `N_TEXT` - 一个代码符号。该符号的值是代码地址，链接程序在合并二进制目标文件时会更新其值。
- ♦ `N_DATA` - 一个数据符号；与 `N_TEXT` 类似，但是用于数据地址。对应代码和数据符号的值不是文件的偏移值而是地址；为了找出文件的偏移，就有必要确定相关部分开始加载的地址并减去它，然后加上该部分的偏移。
- ♦ `N_BSS` - 一个 `BSS` 符号；与代码或数据符号类似，但在二进制目标文件中没有对应的偏移。
- ♦ `N_FN` - 一个文件名符号。在合并二进制目标文件时，链接程序会将该符号插入在二进制文件中的符号之前。符号的名称就是给予链接程序的文件名，而其值是二进制文件中首个代码段地址。

链接和加载时不需要文件名符号，但对于调式程序非常有用。

- **N_STAB** - 屏蔽码用于选择符号调式程序(例如 **gdb**)感兴趣的位；其值在 **stab()**中说明。

n_other - 该字段按照 **n_type** 确定的段，提供有关符号重定位操作的符号独立性信息。目前，**n_other** 字段的最低 4 位含有两个值之一：**AUX_FUNC** 和 **AUX_OBJECT**（有关定义参见<link.h>）。**AUX_FUNC** 将符号与可调用的函数相关，**AUX_OBJECT** 将符号与数据相关，而不管它们是位于代码段还是数据段。该字段主要用于链接程序 **ld**，用于动态可执行程序创建。

n_desc - 保留给调式程序使用；链接程序不对其进行处理。不同的调试程序将该字段用作不同的用途。

n_value - 含有符号的值。对于代码、数据和 **BSS** 符号，这是一个地址；对于其他符号（例如调式程序符号），值可以是任意的。

字符串表是由长度为 **u_int32_t** 后跟一 **null** 结尾的符号字符串组成。长度代表整个表的字节大小，所以在 32 位的机器上其最小值（或者是第 1 个字符串的偏移）总是 4。

12.17 stat.c 程序

12.17.1 功能描述

该程序实现取文件状态信息系统调用 **stat()**和 **fstat()**，并将信息存放在用户的文件状态结构缓冲区中。**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。

12.17.2 代码注释

程序 12-16 linux/fs/stat.c

```

1 /*
2  *  linux/fs/stat.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。
8 #include <sys/stat.h>      // 文件状态头文件。含有文件状态结构 stat {} 和常量。
9
10 #include <linux/fs.h>      // 文件系统头文件。定义文件表结构 (file、m_inode) 等。
11 #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14
15 // 复制文件状态信息。
16 // 参数 inode 是文件 i 节点，statbuf 是用户数据空间中 stat 文件状态结构指针，用于存放取
17 // 得的状态信息。
18 static void cp_stat(struct m_inode * inode, struct stat * statbuf)
19 {
20     struct stat tmp;
21     int i;
22
23     // 首先验证(或分配)存放数据的内存空间。然后临时复制相应节点上的信息。

```

```

20     verify_area(statbuf, sizeof (* statbuf));
21     tmp.st_dev = inode->i_dev;           // 文件所在的设备号。
22     tmp.st_ino = inode->i_num;           // 文件 i 节点号。
23     tmp.st_mode = inode->i_mode;         // 文件属性。
24     tmp.st_nlink = inode->i_nlinks;      // 文件的连接数。
25     tmp.st_uid = inode->i_uid;           // 文件的用户 id。
26     tmp.st_gid = inode->i_gid;           // 文件的组 id。
27     tmp.st_rdev = inode->i_zone[0];      // 设备号（若是特殊字符文件或块设备文件）。
28     tmp.st_size = inode->i_size;         // 文件字节长度（如果文件是常规文件）。
29     tmp.st_atime = inode->i_atime;       // 最后访问时间。
30     tmp.st_mtime = inode->i_mtime;      // 最后修改时间。
31     tmp.st_ctime = inode->i_ctime;      // 最后 i 节点修改时间。
// 最后将这些状态信息复制到用户缓冲区中。
32     for (i=0 ; i<sizeof (tmp) ; i++)
33         put_fs_byte(((char *) &tmp)[i], &((char *) statbuf)[i]);
34 }
35
//// 文件状态系统调用。
// 根据给定的文件名获取相关文件状态信息。
// 参数 filename 是指定的文件名，statbuf 是存放状态信息的缓冲区指针。
// 返回：成功返回 0，若出错则返回出错码。
36 int sys_stat(char * filename, struct stat * statbuf)
37 {
38     struct m_inode * inode;
39
// 首先根据文件名找出对应的 i 节点。然后将 i 节点上的文件状态信息复制到用户缓冲区中，
// 并放回该 i 节点。
40     if (!(inode=namei(filename)))
41         return -ENOENT;
42     cp_stat(inode, statbuf);
43     iput(inode);
44     return 0;
45 }
46
//// 文件状态系统调用。
// 根据给定的文件句柄获取相关文件状态信息。
// 参数 fd 是指定文件的句柄(描述符)，statbuf 是存放状态信息的缓冲区指针。
// 返回：成功返回 0，若出错则返回出错码。
47 int sys_fstat(unsigned int fd, struct stat * statbuf)
48 {
49     struct file * f;
50     struct m_inode * inode;
51
// 首先取文件句柄对应的文件结构，然后从中得到文件的 i 节点。然后将 i 节点上的文件状
// 态信息复制到用户缓冲区中。如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或
// 者该句柄的文件结构指针为空，或者对应文件结构的 i 节点字段为空，则出错，返回出错
// 码并退出。
52     if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
53         return -EBADF;
54     cp_stat(inode, statbuf);
55     return 0;
56 }
57

```

12.18 fcntl.c 程序

12.18.1 功能描述

从本节开始注释的一些文件，都属于对目录和文件进行操作的上层处理程序。

本文件 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() 中指定了新句柄的最小数值，而 dup() 则返回当前值最小的未用句柄。fcntl() 用于修改已打开文件的状态或复制句柄操作。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。本程序中用到的一些常数符号定义在 include/fcntl.h 文件中。建议在阅读本程序时也同时参考该头文件。

函数 dup() 和 dup2() 所返回的新文件句柄与被复制的原句柄将共同使用同一个文件表项。例如当一个进程没有另行打开任何其他文件时，此时若使用 dup() 函数或使用 dup2() 函数但指定新句柄是 3，那么函数执行后的文件句柄示意图见图 12-35 所示。

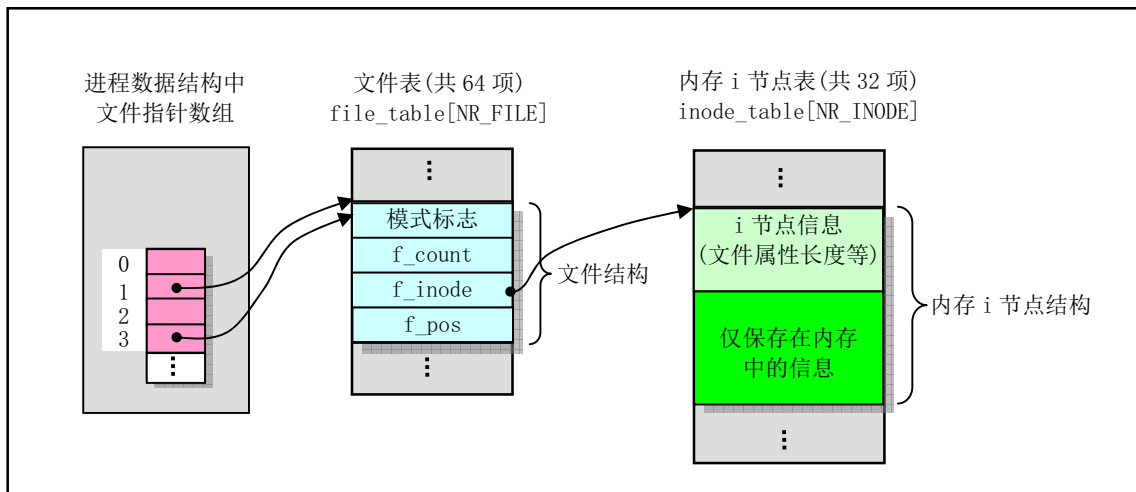


图 12-35 执行 dup(1) 或 dup2(1, 3) 函数后内核中的文件相关结构

另外，由本程序中的 dupfd() 内部函数可以看出，对于使用 dup() 或 dup2() 函数新建的文件句柄，其执行时关闭标志 close_on_exec 会被清除，即在运行 exec() 类函数时不会关闭用 dup() 建立的文件句柄。

由 AT&T 的系统 III 开始采用的 fcntl() 函数主要用于修改已打开文件的属性。它在参数中用控制命令 cmd 在该函数中集成了 4 种功能：

- cmd = F_DUPFD，复制文件句柄。此时 fcntl() 的返回值是新的文件句柄，其值大于等于第 3 个参数指定的值。该新建的文件句柄将与原句柄共同使用同一个文件表项，但其执行时关闭标志被复位。对于该命令，函数 dup(fd) 就等效于 fcntl(fd, F_DUPFD, 0)；而函数 dup2(fd, newfd) 则等效于语句 “close(newfd); fcntl(fd, F_DUPFD, newfd);”

- cmd = F_GETFD 或 F_SETFD，读取或设置文件句柄执行时关闭标志 close_on_exec。在设置该标志时，函数第 3 个参数是该标志的新值。

- cmd = F_GETFL 或 F_SETFL，读取或设置文件操作和访问标志。这些标志有 RDONLY、O_WRONLY、O_RDWR、O_APPEND 和 O_NONBLOCK，它们的具体含义请参见 include/fcntl.h 文件。在设置操作时函数第 3 个参数是文件操作和访问标志的新值，并且只能改动 O_APPEND 和 O_NONBLOCK 标志。

• cmd = F_GETLK、F_SETLK 或 F_SETLKW，读取或设置文件上锁标志。但在 Linux 0.11 内核中没有实现文件记录上锁功能。

12.18.2 代码注释

程序 12-17 linux/fs/fcntl.c

```

1  /*
2  *  linux/fs/fcntl.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <string.h>      // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8  #include <errno.h>      // 错误号头文件。包含系统中各种出错号。
9  #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
10 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
11 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
12
13 #include <fcntl.h>      // 文件控制头文件。定义文件及其描述符的操作控制常数符号。
14 #include <sys/stat.h>   // 文件状态头文件。含有文件状态结构 stat {} 和常量。
15
16 extern int sys_close(int fd);    // 关闭文件系统调用。(fs/open.c, 192)
17
18     // 复制文件句柄（文件描述符）。
19     // 参数 fd 是欲复制的文件句柄，arg 指定新文件句柄的最小数值。
20     // 返回新文件句柄或出错码。
21 static int dupfd(unsigned int fd, unsigned int arg)
22 {
23     // 首先检查函数参数的有效性。如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或者
24     // 该句柄的文件结构不存在，则返回出错码并退出。如果指定的新句柄值 arg 大于最多打开文
25     // 件数，也返回出错码并退出。注意，实际上文件句柄就是进程文件结构指针数组项索引号。
26     if (fd >= NR_OPEN || !current->filp[fd])
27         return -EBADF;
28     if (arg >= NR_OPEN)
29         return -EINVAL;
30     // 然后在当前进程的文件结构指针数组中寻找索引号等于或大于 arg，但还没有使用的项。若
31     // 找到的新句柄值 arg 大于最多打开文件数（即没有空闲项），则返回出错码并退出。
32     while (arg < NR_OPEN)
33         if (current->filp[arg])
34             arg++;
35     else
36         break;
37     if (arg >= NR_OPEN)
38         return -EMFILE;
39     // 否则针对找到的空闲项（句柄），在执行时关闭标志位图 close_on_exec 中复位该句柄位。
40     // 即在运行 exec() 类函数时，不会关闭用 dup() 创建的句柄。并令该文件结构指针等于原句
41     // 柄 fd 的指针，并且将文件引用计数增 1。最后返回新的文件句柄 arg。
42     current->close_on_exec &= ~(1<<arg);
43     (current->filp[arg] = current->filp[fd])->f_count++;
44     return arg;
45 }
```

```

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

```

```

70         case F_GETLK:   case F_SETLK:   case F_SETLKW:   // 未实现。
71             return -1;
72         default:
73             return -1;
74     }
75 }
76

```

12.19 ioctl.c 程序

12.19.1 功能描述

ioctl.c 文件实现了输入/输出控制系统调用 ioctl()。ioctl() 函数可以看作是各个具体设备驱动程序 ioctl 函数的接口函数。该函数将调用文件句柄指定设备文件的驱动程序中的 IO 控制函数。主要调用 tty 字符设备的 tty_ioctl() 函数，对终端的 I/O 进行控制。在编制用户程序时通常采用 POSIX.1 标准定义的 termios 相关函数来设置 tty 设备属性。参见 include/termios.h 文件最后部分。那些函数（例如 tcflow()）在编译环境的函数库 libc.a 中实现，并且通过系统调用还是执行了本程序中的 ioctl() 函数。

12.19.2 代码注释

程序 12-18 linux/fs/ioctl.c

```

1  /*
2   *  linux/fs/ioctl.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <string.h>           // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8  #include <errno.h>           // 错误号头文件。包含系统中各种出错号。
9  #include <sys/stat.h>        // 文件状态头文件。含有文件状态结构 stat {} 和常量。
10
11 #include <linux/sched.h>      // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
12
13 extern int  tty_ioctl(int dev, int cmd, int arg);    // chr_drv/tty_ioctl.c, 115 行。
14
15 // 定义输入输出控制(ioctl)函数指针类型。
16 typedef int (*ioctl_ptr)(int dev, int cmd, int arg);
17
18 // 取系统中设备种数的宏。
19 #define NRDEVS ((sizeof (ioctl_table))/(sizeof (ioctl_ptr)))
20
21 // ioctl 操作函数指针表。
22 static ioctl_ptr ioctl_table[]={
23     NULL,                /* nodev */
24     NULL,                /* /dev/mem */
25     NULL,                /* /dev/fd */
26     NULL,                /* /dev/hd */
27     tty_ioctl,           /* /dev/ttyx */
28     tty_ioctl,           /* /dev/tty */

```

```




26     NULL,          /* /dev/lp */
27     NULL};        /* named pipes */
28
29
30 // 系统调用函数 - 输入输出控制函数。
31 // 该函数首先判断参数给出的文件描述符是否有效。然后根据对应 i 节点中文件属性判断文件
32 // 类型，并根据具体文件类型调用相关的处理函数。
33 // 参数: fd - 文件描述符; cmd - 命令码; arg - 参数。
34 // 返回: 成功则返回 0, 否则返回出错码。
35 int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
36 {
37     struct file * filp;
38     int dev, mode;
39
40     // 首先判断给出的文件描述符的有效性。如果文件描述符超出可打开的文件数, 或者对应描述
41     // 符的文件结构指针为空, 则返回出错码退出。
42     if (fd >= NR_OPEN || !(filp = current->filp[fd]))
43         return -EBADF;
44     // 否则就取对应文件的属性, 并据此判断文件的类型。如果该文件既不是字符设备文件, 也不
45     // 是块设备文件, 则返回出错码退出。若是字符或块设备文件, 则从文件的 i 节点中取设备号。
46     // 如果设备号大于系统现有的设备数, 则返回出错号。
47     mode = filp->f_inode->i_mode;
48     if (!S_ISCHR(mode) && !S_ISBLK(mode))
49         return -EINVAL;
50     dev = filp->f_inode->i_zone[0];
51     if (MAJOR(dev) >= NRDEVS)
52         return -ENODEV;
53     // 然后根据 IO 控制表 ioctl_table 查得对应设备的 ioctl 函数指针, 并调用该函数。如果该设
54     // 备在 ioctl 函数指针表中没有对应函数, 则返回出错码。
55     if (!ioctl_table[MAJOR(dev)])
56         return -ENOTTY;
57     return ioctl_table[MAJOR(dev)](dev, cmd, arg);
58 }

```

第13章 内存管理(mm)

在 Intel 80x86 体系结构中，Linux 内核的内存管理程序采用了分页管理方式。利用页目录和页表结构处理内核中其他部分代码对内存的申请和释放操作。内存的管理是以内存页面为单位进行的，一个内存页面是指地址连续的 4K 字节物理内存。通过页目录项和页表项，可以寻址和管理指定页面的使用情况。在 Linux 0.11 的内存管理目录中共有三个文件，如列表 13-1 中所示：

列表 13-1 内存管理子目录文件列表

	名称	大小	最后修改时间 (GMT)	说明
	Makefile	813 bytes	1991-12-02 03:21:45	
	memory.c	11223 bytes	1991-12-03 00:48:01	
	page.s	508 bytes	1991-10-02 14:16:30	

其中，page.s 文件比较短，仅包含内存页异常的中断处理过程（int 14）。主要实现了对缺页和页写保护的处理。memory.c 是内存页面管理的核心文件，用于内存的初始化操作、页目录和页表的管理和内核其他部分对内存的申请处理过程。

13.1 总体功能

在 Intel 80X86 CPU 中，程序在寻址过程中使用的是由段和偏移值构成的地址。该地址并不能直接用来寻址物理内存地址，因此被称为虚拟地址。为了能寻址物理内存，就需要一种地址变换机制将虚拟地址映射或变换到物理内存中，这种地址变换机制就是内存管理的主要功能之一（内存管理的另外一个主要功能是内存的寻址保护机制。由于篇幅所限，本章不对其进行讨论）。虚拟地址通过段管理机制首先转换成一种中间地址形式—CPU 32 位的线性地址，然后使用分页管理机制将此线性地址映射到物理地址。

为了弄清 Linux 内核对内存的管理操作方式，我们需要了解内存分页管理的工作原理，了解其寻址的机制。分页管理的目的是将物理内存页面映射到某一线性地址处。在分析本章的内存管理程序时，需明确区分清楚给定的地址是指线性地址还是实际物理内存的地址。

13.1.1 内存分页管理机制

在 Intel 80x86 的系统中，内存分页管理是通过页目录表和内存页表所组成的二级表进行的。见图 13-1 所示。其中页目录表和页表的结构是一样的，表项结构也相同，见下面图 13-4 所示。页目录表中的每个表项（简称页目录项）（4 字节）用来寻址一个页表，而每个页表项（4 字节）用来指定一页物理内存页。因此，当指定了一个页目录项和一个页表项，我们就可以唯一地确定所对应的物理内存页。页目录表占用一页内存，因此最多可以寻址 1024 个页表。而每个页表也同样占用一页内存，因此一个页表可以寻址最多 1024 个物理内存页面。这样在 80386 中，一个页目录表所寻址的所有页表共可以寻址 $1024 \times 1024 \times 4096 = 4\text{G}$ 的内存空间。在 Linux 0.11 内核中，所有进程都使用一个页目录表，而每个进程都有自己的页表。内核代码和数据段长度是 16MB，使用了 4 个页表（即 4 个页目录项）。这 4 个页表直接位于页目录表后面，参见 head.s 程序第 109--125 行。经过分段机制变换，内核代码和数据段位于线性地址空间的头 16MB 范围内，再经过分页机制变换，它被直接一一对应地映射到 16MB 的物理内存上。因此对于内核

段来讲其线性地址就是物理地址。

对于应用进程或内核其他部分来讲，在申请内存时使用的是线性地址。接下来我们就要问了：“那么，一个线性地址如何使用这两个表来映射到一个物理地址上呢？”。为了使用分页机制，一个 32 位的线性地址被分成了三个部分，分别用来指定一个页目录项、一个页表项和对应物理内存页上的偏移地址，从而能间接地寻址到线性地址指定的物理内存位置。见图 13-2 所示。

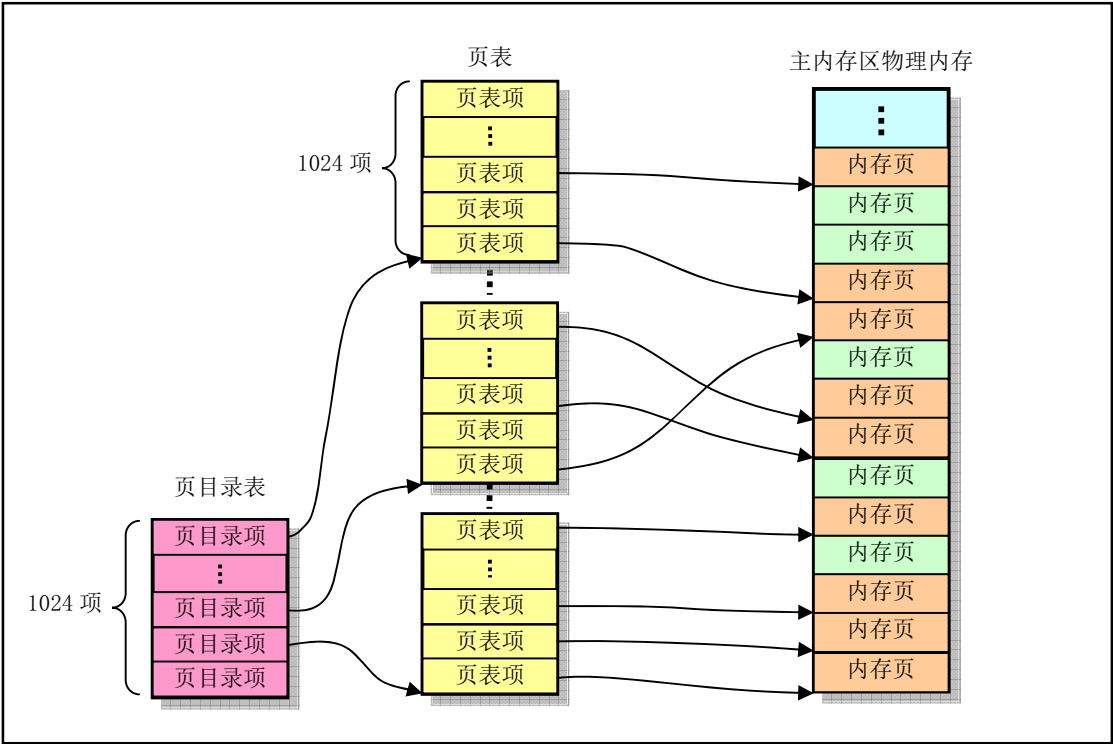


图 13-1 页目录表和页表结构示意图

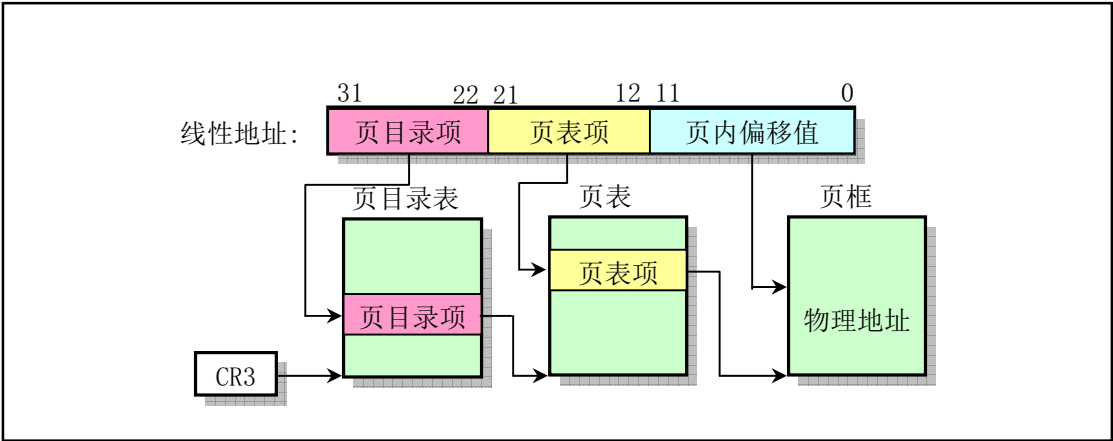


图 13-2 线性地址变换示意图

线性地址的位 31-22 共 10 个比特用来确定页目录中的目录项，位 21-12 用来寻址页目录项指定的页表中的页表项，最后的 12 个比特正好用作页表项指定的一页物理内存中的偏移地址。

在内存管理的函数中，大量使用了从线性地址到实际物理地址的变换计算。对于给定一个进程的线性地址，通过图 13-2 中所示的地址变换关系，我们可以很容易地找到该线性地址对应的页目录项。若该

目录项有效（被使用），则该目录项中的页框地址指定了一个页表在物理内存中的基址，那么结合线性地址中的页表项指针，若该页表项有效，则根据该页表项中的指定的页框地址，我们就可以最终确定指定线性地址对应的实际物理内存页的地址。反之，如果需要一个已知被使用的物理内存页地址，寻找对应的线性地址，则需要对整个页目录表 and 所有页表进行搜索。若该物理内存页被共享，我们就可能会找到多个对应的线性地址来。图 13-3 用形象的方法示出了一个给定的线性地址是如何映射到物理内存页上的。对于第一个进程（任务 0），其页表是在页目录表之后，共 4 页。对于应用程序的进程，其页表所使用的内存是在进程创建时向内存管理程序申请的，因此是在主内存区中。

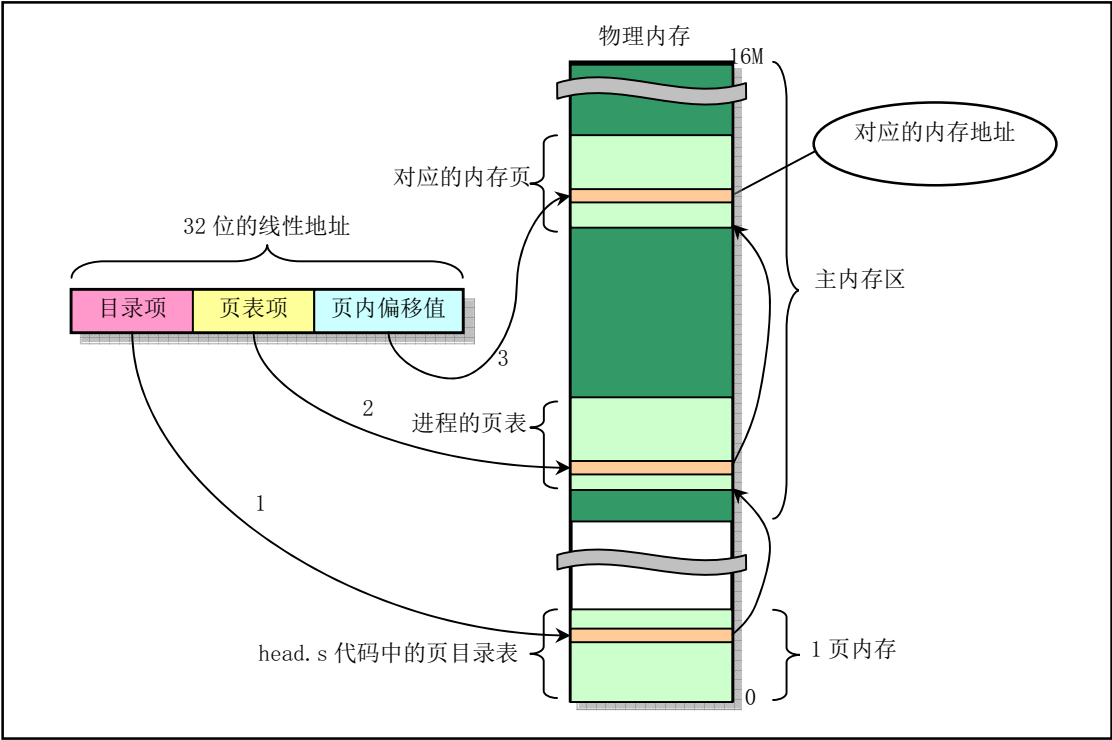


图 13-3 线性地址对应的物理地址

一个系统中可以同时存在多个页目录表，而在某个时刻只有一个页目录表可用。当前的页目录表是用 CPU 的寄存器 CR3 来确定的，它存储着当前页目录表的物理内存地址。但在本书所讨论的 Linux 内核中只使用了一个页目录表。

在图 13-1 中我们看到，每个页表项对应的物理内存页在 4G 的地址范围内是随机的，是由页表项中页框地址内容确定的，也即是由内存管理程序通过设置页表项确定的。每个表项由页框地址、访问标志位、脏（已改写）标志位和存在标志位等构成。表项的结构可参见图 13-4 所示。

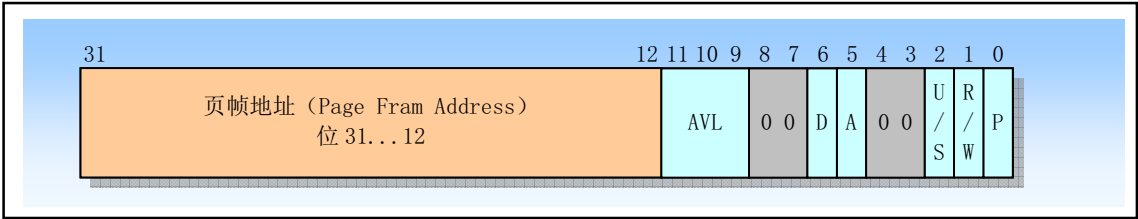


图 13-4 页目录和页表表项结构

其中, 页框地址(PAGE FRAME ADDRESS)指定了一页内存的物理起始地址。因为内存页是位于 4K 边界上的, 所以其低 12 比特总是 0, 因此表项的低 12 比特可作它用。在一个页目录表中, 表项的页框地址是一个页表的起始地址; 在第二级页表中, 页表项的页框地址则包含期望内存操作的物理内存页地址。

图中的存在位 (PRESENT - P) 确定了一个页表项是否可以用于地址转换过程。P=1 表示该项可用。当目录表项或第二级表项的 P=0 时, 则该表项是无效的, 不能用于地址转换过程。此时该表项的所有其他比特位都可供程序使用; 处理器不对这些位进行测试。

当 CPU 试图使用一个页表项进行地址转换时, 如果此时任意一级页表项的 P=0, 则处理器就会发出页异常信号。此时缺页中断异常处理程序就可以把所请求的页加入到物理内存中, 并且导致异常的指令会被重新执行。

已访问 (Accessed - A) 和已修改 (Dirty - D) 比特位用于提供有关页使用的信息。除了页目录项中的已修改位, 这些比特位将由硬件置位, 但不复位。页目录项和页表项的小区别在于页表项有个已写位 D (Dirty), 而页目录项则没有。

在对一页内存进行读或写操作之前, CPU 将设置相关的目录和二级页表项的已访问位。在向一个二级页表项所涵盖的地址进行写操作之前, 处理器将设置该二级页表项的已修改位, 而页目录项中的已修改位是不用的。当所需求的内存超出实际物理内存量时, 内存管理程序就可以使用这些位来确定那些页可以从内存中取走, 以腾出空间。内存管理程序还需负责检测和复位这些比特位。

读/写位 (Read/Write - R/W) 和用户/超级用户位 (User/Supervisor - U/S) 并不用于地址转换, 但用于分页级的保护机制, 是由 CPU 在地址转换过程中同时操作的。

13.1.2 Linux 中物理内存的管理和分配

有了以上概念, 我们就可以说明 Linux 进行内存管理的方法了。但还需要了解一下 Linux 0.11 内核使用内存空间的情况。对于 Linux 0.11 内核, 它默认最多支持 16M 物理内存。在一个具有 16MB 内存的 80x86 计算机系统中, Linux 内核占用物理内存最前段的一部分, 图中 end 标示出内核模块结束的位置。随后是高速缓冲区, 它的最高内存地址为 4M。高速缓冲区被显示内存和 ROM BIOS 分成两段。剩余的内存部分称为主内存区。主内存区就是由本章的程序进行分配管理的。若系统中还存在 RAM 虚拟盘时, 则主内存区前段还要扣除虚拟盘所占的内存空间。当需要使用主内存区时就需要向本章的内存管理程序申请, 所申请的基本单位是内存页。整个物理内存各部分的功能示意图如图 13-5 所示。

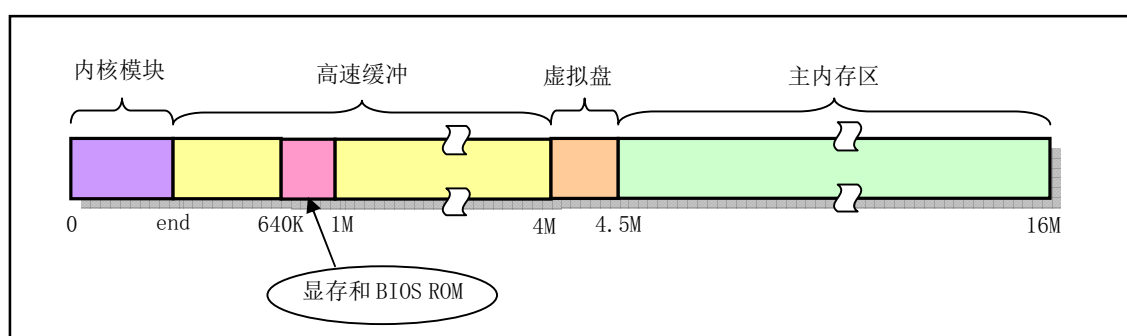


图 13-5 主内存区域示意图

在启动引导一章中, 我们已经知道, Linux 的页目录和页表是在程序 head.s 中设置的。head.s 程序在物理地址 0 处存放了一个页目录表, 紧随其后是 4 个页表。这 4 个页表将被用于内核所占内存区域的影射操作。由于任务 0 的代码和数据包含在内核区域中, 因此任务 0 也使用这些页表。其他的派生进程将在主内存区申请内存页来存放自己的页表。本章中的两个程序就是用于对这些表进行管理操作, 从而实

现对主内存区中内存页面的分配使用。

为了节约物理内存，在调用 `fork()` 生成新进程时，新进程与原进程会共享同一内存区。只有当其中一个进程进行写操作时，系统才会为其另外分配内存页面。这就是写时复制的概念。

`page.s` 程序用于实现页异常中断处理过程（`int 14`）。该中断处理过程对由于缺页和页写保护引起的中断分别调用 `memory.c` 中的 `do_no_page()` 和 `do_wp_page()` 函数进行处理。`do_no_page()` 会把需要的页面从块设备中取到内存指定位置处。在共享内存页面情况下，`do_wp_page()` 会复制被写的页面（copy on write，写时复制），从而也取消了对页面的共享。

13.1.3 Linux 内核对线性地址空间的使用分配

在阅读本章代码时，我们还需要了解一个执行程序进程的代码和数据在其逻辑地址空间中的分布情况，参见下面图 5-12 所示。

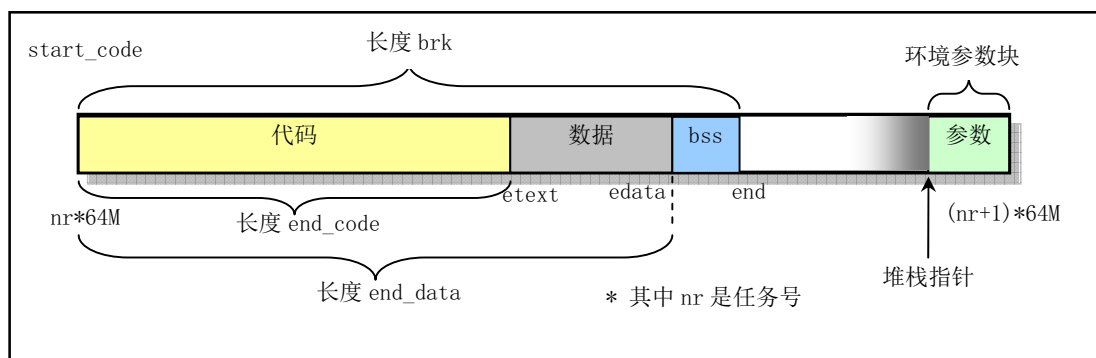


图 13-6 进程代码和数据在其逻辑地址空间中的分布

每个进程在线性地址中都是从 $nr \times 64\text{MB}$ 的地址位置开始（ nr 是任务号），占用逻辑地址空间的范围是 64MB （当然也是线性地址空间的范围）。其中最后部的环境参数数据块最长为 128K ，其左面是起始堆栈指针。另外，图中 `bss` 是进程未初始化的数据段，在进程创建时 `bss` 段的第一页会被初始化为全 0。

13.1.4 页面出错异常处理

在运行于开启了分页机制（`PG=1`）的状态下，若 CPU 在执行线性地址变换到物理地址的过程中检测到以下条件，就会引起页出错异常中断 `int 14`：

- 地址变换过程中用到的页目录项或页表项中存在位（`P`）等于 0；
- 当前执行程序没有足够的特权访问指定的页面。

此时 CPU 会向页出错异常处理程序提供以下两方面信息来协助诊断和纠正错误：

- 栈中的一个出错码（error code）。出错码的格式是一个 32 位的长字。但只有最低 3 个比特有用，它们的名称与页表项中的最后三位相同（`U/S`、`W/R`、`P`）。它们的含义和作用分别是：
 - ◆ 位 0（`P`），异常是由于页面不存在或违反访问特权而引发。`P=0`，表示页不存在；`P=1` 表示违反页级保护权限。
 - ◆ 位 1（`W/R`），异常是由于内存读或写操作引起。`W/R=0`，表示由读操作引起；`W/R=1`，表示由写操作引起。
 - ◆ 位 2（`U/S`），发生异常时 CPU 执行的代码级别。`U/S=0`，表示 CPU 正在执行超级用户代码；`U/S=1`，表示 CPU 正在执行一般用户代码。
- 在控制寄存器 `CR2` 中的线性地址。CPU 会把引起异常的访问使用的线性地址存放在 `CR2` 中。页出错异常处理程序可以使用这个地址来定位相关的页目录和页表项。

后面将要描述的 `page.s` 程序就是利用以上信息来区分是缺页异常还是写保护异常，从而确定调用

memory.c 程序中的缺页处理函数 `do_no_page()`或写保护函数 `do_wp_page()`函数。

13.1.5 写时复制（copy on write）机制

写时复制是一种推迟或免除复制数据的一种方法。此时内核并不去复制进程整个地址空间中的数据，而是让父进程和子进程共享同一个拷贝。当进程 A 使用系统调用 `fork` 创建出一个子进程 B 时，由于子进程 B 实际上是父进程 A 的一个拷贝，因此会拥有与父进程相同的物理页面。也即为了达到节约内存和加快创建进程速度的目标，`fork()`函数会让子进程 B 以只读方式共享父进程 A 的物理页面。同时将父进程 A 对这些物理页面的访问权限也设成只读（详见 memory.c 程序中的 `copy_page_tables()`函数）。这样一来，当父进程 A 或子进程 B 任何一方对这些共享物理页面执行写操作时，都会产生页面出错异常（`page_fault int14`）中断，此时 CPU 就会执行系统提供的异常处理函数 `do_wp_page()`来试图解决这个异常。这就是写时复制机制。

`do_wp_page()`会对这块导致写入异常中断的物理页面进行取消共享操作（使用 `un_wp_page()`函数），并为写进程复制一新的物理页面，使父进程 A 和子进程 B 各自拥有一块内容相同的物理页面。这时才真正地进行了复制操作（只复制这一块物理页面）。并且把将要执行写入操作的这块物理页面标记成可以写访问的。最后，从异常处理函数中返回时，CPU 就会重新执行刚才导致异常的写入操作指令，使进程能够继续执行下去。

因此，对于进程在自己的虚拟地址范围内进行写操作时，就会使用上面这种被动的写时复制操作，也即：写操作 -> 页面异常中断 -> 处理写保护异常 -> 重新执行写操作指令。而对于系统内核代码，当在某个进程的虚拟地址范围内执行写操作时，例如进程调用某个系统调用，若该系统调用会将数据复制到进程的缓冲区域中，则内核会通过 `verify_area()`函数首先主动地调用内存页面验证函数 `write_verify()`，来判断是否有页面共享的情况存在，如果有，就进行页面的写时复制操作。

另外，值得注意的一点是在 Linux 0.11 内核中，在内核代码地址空间（线性地址<1MB）执行 `fork()`来创建进程使并没有采用写时复制技术。因此当进程 0（idle 进程）在内核空间创建进程 1（init 进程）时将使用同一段代码和数据段。但由于进程 1 复制的页表项也是只读的，因此当进程 1 需要执行堆栈（写）操作时也会引起页面异常，从而在这种情况下内存管理程序也会在主内存区中为该进程分配内存。

由此可见，写时复制把对内存页面的复制操作推迟到实际要进行写操作的时刻，在页面不会被写的情况下就可以根本不用进行页面复制操作。例如，当 `fork()`创建了一个进程后立即调用 `execve()`去执行一个新程序的时候。因此这种技术可以避免不必要的内存页面复制的开销。

13.1.6 需求加载（Load on demand）机制

在使用 `execve()`系统调用加载运行文件系统上的一个执行映像文件时，内核除了在 CPU 的 4G 线性地址空间中为对应进程分配了 64MB 的连续空间，并为其环境参数和命令行参数分配和映射了一定数量的物理内存页面以外，实际上并没有给执行程序分配其它任何物理内存页面。当然也谈不上从文件系统上加载执行映像文件中的代码和数据。因此一旦该程序从设定的入口执行点开始运行就会立刻引起 CPU 产生一个缺页异常（执行指针所在的内存页面不存在）。此时内核的缺页异常处理程序才会根据引起缺页异常的具体线性地址把执行文件中相关的代码页从文件系统中加载到物理内存页面中，并映射到进程逻辑地址中指定的页面位置处。当异常处理程序返回后 CPU 就会重新执行引起异常的指令，使得执行程序能够得以继续执行。若在执行过程中又要运行到另一页中还未加载的代码，或者代码指令需要访问还未加载的数据，那么 CPU 同样会产生一个缺页异常中断，此时内核就又会把执行程序中的其他对应页面内容加载到内存中。就这样，执行文件中只有运行到（用到）的代码或数据页面才会被内核加载到物理内存中。这种仅在实际需要时才加载执行文件中页面的方法被称为需求加载（Load on demand）技术或需求分页（demand-paging）技术。

采用需求加载技术的一个明显优点是在调用 `execve()`系统后能够让执行程序立刻开始运行，而无需等待多次的块设备 I/O 操作把整个执行文件映像加载到内存中后才开始运行。因此系统对执行程序的加

载执行速度将大大地提高。但这种技术对被加载执行目标文件的格式有一定要求。它要求被执行的文件目标格式是 ZMAGIC 类型的，即需求分页格式的目标文件格式。在这种目标文件格式中，程序的代码段和数据段都从页面边界开始存放，以适应内核以一个页面为单位读取代码或数据内容。

13.2 Makefile 文件

13.2.1 功能描述

本文件是 mm 目录中程序的编译管理配置文件，共 make 程序使用。

13.2.2 代码注释

程序 13-1 linux/mm/Makefile

```

1 CC      =gcc      # GNU C 语言编译器。
2 CFLAGS  =-O -Wall -fstrength-reduce -fcombine-regs -fomit-frame-pointer \
3          -finline-functions -nostdinc -I../include
# 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；-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)。
4 AS      =gas      # GNU 的汇编程序。
5 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。
6 LD      =gld      # GNU 的连接程序。
7 CPP     =gcc -E -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。
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。
9 .c.o:
10      $(CC) $(CFLAGS) \
11      -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。22 行是实现该操作的具体命令。
12 .s.o:
13      $(AS) -o $.o $<
14 .c.s:      # 类似上面，*.c 文件→*.s 汇编程序文件。不进行连接。
15      $(CC) $(CFLAGS) \
16      -S -o $.s $<
17
18 OBJJS     = memory.o page.o  # 定义目标文件变量 OBJJS。
19
20 all: mm.o
21
# 在有了先决条件 OBJJS 后使用下面的命令连接成目标 mm.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 mm.o: $(OBJS)
23     $(LD) -r -o mm.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 mm/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 memory.o : memory.c ../include/signal.h ../include/sys/types.h \
36     ../include/asm/system.h ../include/linux/sched.h ../include/linux/head.h \
37     ../include/linux/fs.h ../include/linux/mm.h ../include/linux/kernel.h

```

13.3 memory.c 程序

13.3.1 功能描述

本程序进行内存分页的管理。实现了对主内存区内存页面的动态分配和回收操作。对于内核代码和数据所占物理内存区域以外的内存（1MB 以上内存区域），内核使用了一个字节数组 `mem_map[]` 来表示物理内存页面的状态。每个字节描述一个物理内存页的占用状态。其中的值表示被占用的次数，0 表示对应的物理内存空闲着。当申请一页物理内存时，就将对应字节的值增 1。

在内存管理初始化过程中，系统首先计算出 1MB 以上内存区域对于的内存页面数（`PAGING_PAGES`），并把 `mem_map[]` 所有项都置为 100（占用），然后把主内存区域对应的 `mem_map[]` 项中的值清零。因此内核所使用的位于 1MB 地址以上的高速缓冲区域以及虚拟磁盘区域（若有的话）都已经被初始化成占用状态。`mem_map[]` 中对应主内存区域的项则在系统使用过程中进行设置或复位。例如，对于图 13-5 所示的具有 16MB 物理内存并设置了 512KB 虚拟磁盘的机器，`mem_map[]` 数组共有 $(16\text{MB} - 1\text{MB})/4\text{KB} = 3840$ 项，即对应 3840 个页面。其中主内存区拥有的页面数为 $(16\text{MB} - 4.5\text{MB})/4\text{KB} = 2944$ 个，对应 `mem_map[]` 数组的最后 2944 项，而前 896 项则对应 1MB 以上的高速缓冲区和虚拟磁盘所占有的物理页面。因此在内存管理初始化过程中，`mem_map[]` 的前 896 项被设置为占用状态（值为 100），不可再被分配使用。而后 2944 项的值被清 0，可被内存管理程序分配使用。参见图 13-7 所示。

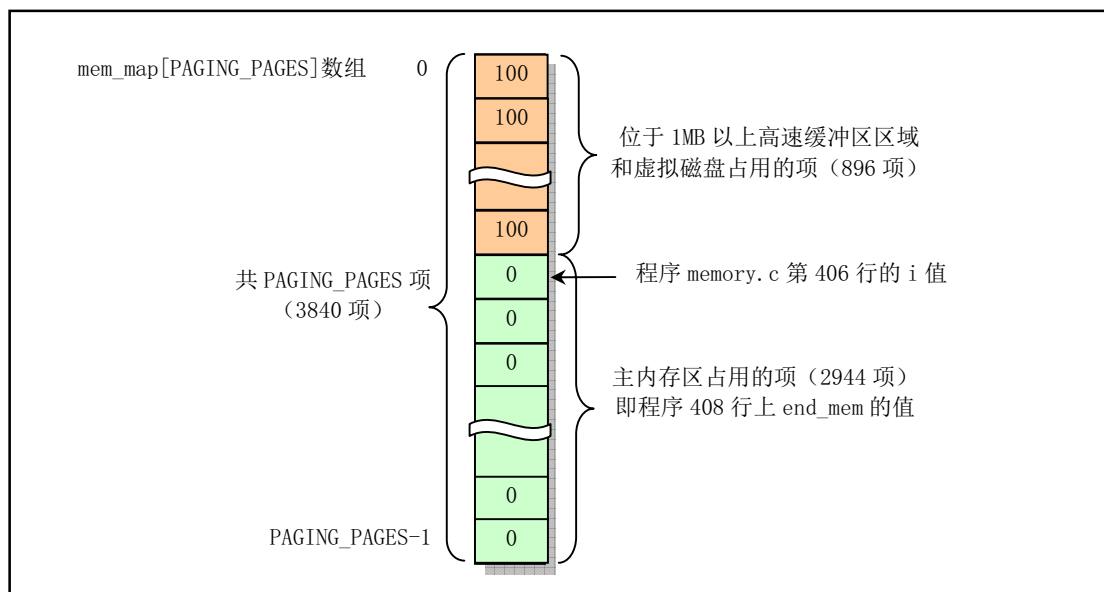


图 13-7 具有 16MB 物理内存和 512KB 虚拟磁盘区机器的 mem_map[] 数组初始化情况

对于进程虚拟地址（或逻辑地址）的管理，内核使用了处理器的页目录表和页表结构来管理。而物理内存页与线性地址之间的映射关系则是通过修改页目录和页表项的内容来处理。下面对程序中所提供的几个主要函数进行详细说明。

get_free_page()和 free_page()这两个函数是专门用来管理主内存区中物理内存的占用和空闲情况，与每个进程的线性地址无关。

get_free_page()函数用于在主内存区中申请一页空闲内存页，并返回物理内存页的起始地址。它首先扫描内存页面字节图数组 mem_map[]，寻找值是 0 的字节项（对应空闲页面）。若无则返回 0 结束，表示物理内存已使用完。若找到值为 0 的字节，则将其置 1，并换算出对应空闲页面的起始地址。然后对该内存页面作清零操作。最后返回该空闲页面的物理内存起始地址。

free_page()用于释放指定地址处的一页物理内存。它首先判断指定的内存地址是否<1M，若是则返回，因为 1M 以内是内核专用的；若指定的物理内存地址大于或等于实际内存最高端地址，则显示出错信息；然后由指定的内存地址换算出页面号: (addr - 1M)/4K；接着判断页面号对应的 mem_map[] 字节项是否为 0，若不为 0，则减 1 返回；否则对该字节项清零，并显示“试图释放一空闲页面”的出错信息。

free_page_tables()和 copy_page_tables()这两个函数则以一个页表对应的物理内存块（4M）为单位，释放或复制指定线性地址和长度（页表个数）对应的物理内存页块。不仅对管理线性地址的页目录和页表中的对应项内容进行修改，而且也对每个页表中所有页表项对应的物理内存页进行释放或占用操作。

free_page_tables()用于释放指定线性地址和长度（页表个数）对应的物理内存页。它首先判断指定的线性地址是否在 4M 的边界上，若不是则显示出错信息，并死机；然后判断指定的地址值是否=0，若是，则显示出错信息“试图释放内核和缓冲区所占用的空间”，并死机；接着计算在页目录表中所占用的目录项数 size，也即页表个数，并计算对应的起始目录项号；然后从对应起始目录项开始，释放所占用的所有 size 个目录项；同时释放对应目录项所指的页表中的所有页表项和相应的物理内存页；最后刷新页变换高速缓冲。

copy_page_tables()用于复制指定线性地址和长度（页表个数）内存对应的页目录项和页表，从而被复制的页目录和页表对应的原物理内存区被共享使用。该函数首先验证指定的源线性地址和目的线性地址是否都在 4Mb 的内存边界地址上，否则就显示出错信息，并死机；然后由指定线性地址换算出对应的起始页目录项 (from_dir, to_dir)；并计算需复制的内存区占用的页表数（即页目录项数）；接着开始分别将原目录项和页表项复制到新的空闲目录项和页表项中。页目录表只有一个，而新进程的页表需要申请

空闲内存页面来存放；此后再将原始和新的页目录和页表项都设置成只读的页面。当有写操作时就利用页异常中断调用，执行写时复制操作。最后对共享物理内存页对应的字节图数组 `mem_map[]` 的标志进行增 1 操作。

`put_page()` 用于将一指定的物理内存页面映射到指定的线性地址处。它首先判断指定的内存页面地址的有效性，要在 1M 和系统最高端内存地址之外，否则发出警告；然后计算该指定线性地址在页目录表中对应的目录项；此时若该目录项有效 ($P=1$)，则取其对应页表的地址；否则申请空闲页给页表使用，并设置该页表中对应页表项的属性。最后仍返回指定的物理内存页面地址。

`do_wp_page()` 是页异常中断过程（在 `mm/page.s` 中实现）中调用的页写保护处理函数。它首先判断地址是否在进程的代码区域，若是则终止程序（代码不能被改动）；然后执行写时复制页面的操作（Copy on Write）。

`do_no_page()` 是页异常中断过程中调用的缺页处理函数。它首先判断指定的线性地址在一个进程空间中相对于进程基址的偏移长度值。如果它大于代码加数据长度，或者进程刚开始创建，则立刻申请一页物理内存，并映射到进程线性地址中，然后返回；接着尝试进行页面共享操作，若成功，则立刻返回；否则申请一页内存并从设备中读入一页信息；若加入该页信息时，指定线性地址+1 页长度超过了进程代码加数据的长度，则将超过的部分清零。然后将该页映射到指定的线性地址处。

`get_empty_page()` 用于取得一页空闲物理内存并映射到指定线性地址处。主要使用了 `get_free_page()` 和 `put_page()` 函数来实现该功能。

13.3.2 代码注释

程序 13-2 linux/mm/memory.c

```

1  /*
2   *  linux/mm/memory.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  demand-loading started 01.12.91 - seems it is high on the list of
9   *  things wanted, and it should be easy to implement. - Linus
10  */
11  /*
12   * 需求加载是从 01.12.91 开始编写的 - 在程序编制表中似乎是最重要的程序，
13   * 并且应该是很容易编制的 - Linus
14   */
15
16  /*
17   *  Ok, demand-loading was easy, shared pages a little bit trickier. Shared
18   *  pages started 02.12.91, seems to work. - Linus.
19   *
20   *  Tested sharing by executing about 30 /bin/sh: under the old kernel it
21   *  would have taken more than the 6M I have free, but it worked well as
22   *  far as I could see.
23   *
24   *  Also corrected some "invalidate()"s - I wasn't doing enough of them.
25   */
26  /*
27   *  OK, 需求加载是比较容易编写的，而共享页面却需要有点技巧。共享页面程序是
28   *  02.12.91 开始编写的，好象能够工作 - Linus。

```

```

*
* 通过执行大约 30 个/bin/sh 对共享操作进行了测试：在老内核当中需要占用多于
* 6M 的内存，而目前却不用。现在看来工作得很好。
*
* 对“invalidate()”函数也进行了修正 - 在这方面我还做的不够。
*/

22
23 #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。
24
25 #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。
26
27 #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。
28 #include <linux/head.h>      // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
29 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原型定义。
30
    // 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。
31 volatile void do_exit(long code); // 进程退出处理函数，在 kernel/exit.c，102 行。
32
    //// 显示内存已用完出错信息，并退出。
33 static inline volatile void oom(void)
34 {
35     printk("out of memory\n");
36     do_exit(SIGSEGV);          // do_exit() 应该使用退出代码，这里用了信号值 SIGSEGV(11)
37 }                               // 相同值的出错码含义是“资源暂时不可用”，正好同义。
38
    // 刷新页变换高速缓冲宏函数。
    // 为了提高地址转换的效率，CPU 将最近使用的页表数据存放在芯片中高速缓冲中。在修改过
    // 页表信息之后，就需要刷新该缓冲区。这里使用重新加载页目录基址寄存器 cr3 的方法来
    // 进行刷新。下面 eax = 0，是页目录的基址。
39 #define invalidate() \
40 __asm__ ("movl %%eax, %%cr3::: \"a\" (0)")
41
42 /* these are not to be changed without changing head.s etc */
    /* 下面定义若需要改动，则需要与 head.s 等文件中的相关信息一起改变 */
    // Linux 0.11 内核默认支持的最大内存容量是 16MB，可以修改这些定义以适合更多的内存。
43 #define LOW_MEM 0x100000      // 内存低端（1MB）。
44 #define PAGING_MEMORY (15*1024*1024) // 分页内存 15MB。主内存区最多 15M。
45 #define PAGING_PAGES (PAGING_MEMORY>>12) // 分页后的物理内存页面数（3840）。
46 #define MAP_NR(addr) (((addr)-LOW_MEM)>>12) // 指定内存地址映射为页号。
47 #define USED 100              // 页面被占用标志，参见 405 行。
48
    // CODE_SPACE(addr) (((addr)+0xfff)&~0xfff)<current->start_code+current->end_code)。
    // 该宏用于判断给定线性地址是否位于当前进程的代码段中，“((addr)+4095)&~4095”用于
    // 取得线性地址 addr 所在内存页面的末端地址。参见 252 行。
49 #define CODE_SPACE(addr) (((addr)+4095)&~4095) < \
50 current->start_code + current->end_code)
51
52 static long HIGH_MEMORY = 0;          // 全局变量，存放实际物理内存最高端地址。
53
    // 从 from 处复制 1 页内存到 to 处（4K 字节）。
54 #define copy_page(from,to) \

```

```

55 __asm__( "cld ; rep ; movsl":: "S" (from), "D" (to), "c" (1024): "cx", "di", "si")
56
    // 物理内存映射字节图（1 字节代表 1 页内存）。每个页面对应的字节用于标志页面当前被引用
    // （占用）次数。它最大可以映射 15Mb 的内存空间。在初始化函数 mem_init() 中，对于不能用
    // 作主内存区页面的位置均都预先被设置成 USED（100）。
57 static unsigned char mem_map [ PAGING_PAGES ] = {0,};
58
59 /*
60  * Get physical address of first (actually last :-) free page, and mark it
61  * used. If no free pages left, return 0.
62  */
    /*
    * 获取首个（实际上是最后 1 个:-）空闲页面，并标记为已使用。如果没有空闲页面，
    * 就返回 0。
    */
    // 在主内存区中取空闲物理页面。如果已经没有可用物理内存页面，则返回 0。
    // 输入：%1(ax=0) - 0；%2(LOW_MEM) 内存字节位图管理的起始位置；%3(cx= PAGING_PAGES)；
    // %4(edi=mem_map+PAGING_PAGES-1)。
    // 输出：返回%0 (ax = 物理页面起始地址)。
    // 上面%4 寄存器实际指向 mem_map[] 内存字节位图的最后一个字节。本函数从位图末端开始向
    // 前扫描所有页面标志（页面总数为 PAGING_PAGES），若有页面空闲（内存位图字节为 0）则
    // 返回页面地址。注意！本函数只是指出在主内存区的一页空闲物理页面，但并没有映射到某
    // 个进程的地址空间中去。后面的 put_page() 函数即用于把指定页面映射到某个进程的地址
    // 空间中。当然对于内核使用本函数并不需要再使用 put_page() 进行映射，因为内核代码和
    // 数据空间（16MB）已经对等地映射到物理地址空间。
    // 第 65 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中，以便于高效访问和
    // 操作。这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册“在指定寄存
    // 器中的变量”。
63 unsigned long get_free_page(void)
64 {
65     register unsigned long __res asm("ax");
66
67     __asm__( "std ; repne ; scasb\n\t" // 置方向位，al(0) 与对应每个页面的(di)内容比较，
68             "jne 1f\n\t" // 如果没有等于 0 的字节，则跳转结束（返回 0）。
69             "movb $1,1(%edi)\n\t" // 1 =>[1+edi]，将对应页面内存映像比特位置 1。
70             "sall $12,%ecx\n\t" // 页面数*4K = 相对页面起始地址。
71             "addl %2,%ecx\n\t" // 再加上低端内存地址，得页面实际物理起始地址。
72             "movl %%ecx,%%edx\n\t" // 将页面实际起始地址→edx 寄存器。
73             "movl $1024,%ecx\n\t" // 寄存器 ecx 置计数值 1024。
74             "leal 4092(%%edx),%%edi\n\t" // 将 4092+edx 的位置→edi（该页面的末端）。
75             "rep ; stosl\n\t" // 将 edi 所指内存清零（反方向，即将该页面清零）。
76             "movl %%edx,%%eax\n\t" // 将页面起始地址→eax（返回值）。
77             "1:"
78             : "=a" (__res)
79             : "" (0), "i" (LOW_MEM), "c" (PAGING_PAGES),
80               "D" (mem_map+PAGING_PAGES-1)
81             : "di", "cx", "dx");
82     return __res; // 返回空闲物理页面地址（若无空闲页面则返回 0）。
83 }
84
85 /*
86  * Free a page of memory at physical address 'addr'. Used by
87  * 'free_page_tables()'

```

```

88  */
   /*
    * 释放物理地址'addr'处的一页内存。用于函数'free_page_tables()'。
    */
    //// 释放物理地址 addr 开始的 1 页面内存。
    // 物理地址 1MB 以下的内存空间用于内核程序和缓冲，不作为分配页面的内存空间。因此
    // 参数 addr 需要大于 1MB。
89 void free_page(unsigned long addr)
90 {
    // 首先判断参数给定的物理地址 addr 的合理性。如果物理地址 addr 小于内存低端（1MB），
    // 则表示在内核程序或高速缓冲中，对此不予处理。如果物理地址 addr >= 系统所含物理
    // 内存最高端，则显示出错信息并且内核停止工作。
91     if (addr < LOW_MEM) return;
92     if (addr >= HIGH_MEMORY)
93         panic("trying to free nonexistent page");
    // 如果对参数 addr 验证通过，那么就根据这个物理地址换算出从内存低端开始计起的内存
    // 页面号。页面号 = (addr - LOW_MEM)/4096。可见页面号从 0 号开始计起。此时 addr
    // 中存放着页面号。如果该页面号对应的页面映射字节不等于 0，则减 1 返回。此时该映射
    // 字节值应该为 0，表示页面已释放。如果对应页面字节原本就是 0，表示该物理页面本来
    // 就是空闲的，说明内核代码出问题。于是显示出错信息并停机。
94     addr -= LOW_MEM;
95     addr >>= 12;
96     if (mem_map[addr]-->0) return;
97     mem_map[addr]=0;
98     panic("trying to free free page");
99 }
100
101 /*
102  * This function frees a continuous block of page tables, as needed
103  * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
104  */
   /*
    * 下面函数释放页表连续的内存块，'exit()'需要该函数。与 copy_page_tables()
    * 类似，该函数仅处理 4Mb 长度的内存块。
    */
    //// 根据指定的线性地址和限长（页表个数），释放对应内存页表指定的内存块并置表项空闲。
    // 页目录位于物理地址 0 开始处，共 1024 项，每项 4 字节，共占 4K 字节。每个目录项指定一
    // 个页表。内核页表从物理地址 0x1000 处开始（紧接着目录空间），共 4 个页表。每个页表有
    // 1024 项，每项 4 字节。因此也占 4K（1 页）内存。各进程（除了在内核代码中的进程 0 和 1）
    // 的页表所占据的页面在进程被创建时由内核为其在主内存区申请得到。每个页表项对应 1 页
    // 物理内存，因此一个页表最多可映射 4MB 的物理内存。
    // 参数：from - 起始线性基地址；size - 释放的字节长度。
105 int free_page_tables(unsigned long from, unsigned long size)
106 {
107     unsigned long *pg_table;
108     unsigned long *dir, nr;
109
    // 首先检测参数 from 给出的线性基地址是否在 4MB 的边界处。因为该函数只能处理这种情况。
    // 若 from = 0，则出错。说明试图释放内核和缓冲所占空间。
110     if (from & 0x3ffff)
111         panic("free_page_tables called with wrong alignment");
112     if (!from)
113         panic("Trying to free up swapper memory space");

```

```

// 然后计算参数 size 给出的长度所占的页目录项数（4MB 的进位整数倍），也即所占页表数。
// 因为 1 个页表可管理 4MB 物理内存，所以这里用右移 22 位的方式把需要复制的内存长度值
// 除以 4MB。其中加上 0x3fffff（即 4Mb -1）用于得到进位整数倍结果，即除操作若有余数
// 则进 1。例如，如果原 size = 4.01Mb，那么可得到结果 size = 2。接着计算给出的线性
// 地址对应的起始目录项。对应的目录项号 = from >> 22。因为每项占 4 字节，并且由于
// 页目录表从物理地址 0 开始存放，因此实际目录项指针 = 目录项号<<2，也即 (from>>20)。
// “与”上 0xffc 确保目录项指针范围有效，即用于屏蔽目录项指针最后 2 位。因为只移动
// 了 20 位，因此最后 2 位是页表项索引的内容，应屏蔽掉。
114     size = (size + 0x3fffff) >> 22;
115     dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */

// 此时 size 是释放的页表个数，即页目录项数，而 dir 是起始目录项指针。现在开始循环
// 操作页目录项，依次释放每个页表中的页表项。如果当前目录项无效（P 位=0），表示该
// 目录项没有使用（对应的页表不存在），则继续处理下一个目录项。否则从目录项中取出
// 页表地址 pg_table，并对该页表中的 1024 个表项进行处理，释放有效页表项（P 位=1）
// 对应的物理内存页面。然后把该页表项清零，并继续处理下一页表项。当一个页表所有
// 表项都处理完毕就释放该页表自身占据的内存页面，并继续处理下一页目录项。最后刷新
// 页变换高速缓冲，并返回 0。
116     for ( ; size-->0 ; dir++) {
117         if (!(1 & *dir))
118             continue;
119         pg_table = (unsigned long *) (0xfffff000 & *dir); // 取页表地址。
120         for (nr=0 ; nr<1024 ; nr++) {
121             if (1 & *pg_table) // 若该项有效，则释放对应页。
122                 free_page(0xfffff000 & *pg_table);
123             *pg_table = 0; // 该页表项内容清零。
124             pg_table++; // 指向页表中下一项。
125         }
126         free_page(0xfffff000 & *dir); // 释放该页表所占内存页面。
127         *dir = 0; // 对应页表的目录项清零。
128     }
129     invalidate(); // 刷新页变换高速缓冲。
130     return 0;
131 }
132
133 /*
134  * Well, here is one of the most complicated functions in mm. It
135  * copies a range of linear addresses by copying only the pages.
136  * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
137  *
138  * Note! We don't copy just any chunks of memory - addresses have to
139  * be divisible by 4Mb (one page-directory entry), as this makes the
140  * function easier. It's used only by fork anyway.
141  *
142  * NOTE 2!! When from==0 we are copying kernel space for the first
143  * fork(). Then we DONT want to copy a full page-directory entry, as
144  * that would lead to some serious memory waste - we just copy the
145  * first 160 pages - 640kB. Even that is more than we need, but it
146  * doesn't take any more memory - we don't copy-on-write in the low
147  * 1 Mb-range, so the pages can be shared with the kernel. Thus the
148  * special case for nr=xxxx.
149  */
150 /*

```



```

* 好了，下面是内存管理 mm 中最为复杂的程序之一。它通过只复制内存页面
* 来拷贝一定范围内线性地址中的内容。希望代码中没有错误，因为我不想
* 再调试这块代码了:-)。
*
* 注意！我们并不复制任何内存块 - 内存块的地址需要是 4Mb 的倍数（正好
* 一个页目录项对应的内存长度），因为这样处理可使函数很简单。不管怎
* 样，它仅被 fork() 使用。
*
* 注意 2！！ 当 from==0 时，说明是在为第一次 fork() 调用复制内核空间。
* 此时我们就不想复制整个页目录项对应的内存，因为这样做会导致内存严
* 重浪费 - 我们只须复制开头 160 个页面 - 对应 640kB。即使是复制这些
* 页面也已经超出我们的需求，但这不会占用更多的内存 - 在低 1Mb 内存
* 范围内我们不执行写时复制操作，所以这些页面可以与内核共享。因此这
* 是 nr=xxxx 的特殊情况（nr 在程序中指页面数）。
*/
///// 复制页目录表项和页表项。
// 复制指定线性地址和长度内存对应的页目录项和页表项，从而被复制的页目录和页表对应
// 的原物理内存页面区被两套页表映射而共享使用。复制时，需申请新页面来存放新页表，
// 原物理内存区将被共享。此后两个进程（父进程和其子进程）将共享内存区，直到有一个
// 进程执行写操作时，内核才会为写操作进程分配新的内存页（写时复制机制）。
// 参数 from、to 是线性地址，size 是需要复制（共享）的内存长度，单位是字节。
150 int copy_page_tables(unsigned long from, unsigned long to, long size)
151 {
152     unsigned long * from_page_table;
153     unsigned long * to_page_table;
154     unsigned long this_page;
155     unsigned long * from_dir, * to_dir;
156     unsigned long nr;
157
// 首先检测参数给出的源地址 from 和目的地址 to 的有效性。源地址和目的地址都需要在 4Mb
// 内存边界地址上。否则出错死机。作这样的要求是因为一个页表的 1024 项可管理 4Mb 内存。
// 源地址 from 和目的地址 to 只有满足这个要求才能保证从一个页表的第 1 项开始复制页表
// 项，并且新页表的最初所有项都是有效的。然后取得源地址和目的地址的起始目录项指针
// （from_dir 和 to_dir）。再根据参数给出的长度 size 计算要复制的内存块占用的页表数
// （即目录项数）。参见前面对 114、115 行的解释。
158     if ((from & 0x3ffff) || (to & 0x3ffff))
159         panic("copy_page_tables called with wrong alignment");
160     from_dir = (unsigned long *) ((from >> 20) & 0xffc); /* _pg_dir = 0 */
161     to_dir = (unsigned long *) ((to >> 20) & 0xffc);
162     size = ((unsigned) (size + 0x3ffff)) >> 22;
// 在得到了源起始目录项指针 from_dir 和目的起始目录项指针 to_dir 以及需要复制的页表
// 个数 size 后，下面开始对每个页目录项依次申请 1 页内存来保存对应的页表，并且开始
// 页表项复制操作。如果目的目录项指定的页表已经存在（P=1），则出错死机。如果源目
// 录项无效，即指定的页表不存在（P=0），则继续循环处理下一个页目录项。
163     for( ; size-->0 ; from_dir++, to_dir++) {
164         if (1 & *to_dir)
165             panic("copy_page_tables: already exist");
166         if (!(1 & *from_dir))
167             continue;
// 在验证了当前源目录项和目的项正常之后，我们取源目录项中页表地址 from_page_table。
// 为了保存目的目录项对应的页表，需要在主内存区中申请 1 页空闲内存页。如果取空闲页面
// 函数 get_free_page() 返回 0，则说明没有申请到空闲内存页面，可能是内存不够。于是返
// 回-1 值退出。

```

```

168         from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
169         if (!(to_page_table = (unsigned long *) get_free_page()))
170             return -1;          /* Out of memory, see freeing */
// 否则我们设置目的目录项信息，把最后 3 位置位，即当前目的目录项“或”上 7，表示对应
// 页表映射的内存页面是用户级的，并且可读写、存在 (Usr, R/W, Present)。(如果 U/S
// 位是 0，则 R/W 就没有作用。如果 U/S 是 1，而 R/W 是 0，那么运行在用户层的代码就只能
// 读页面。如果 U/S 和 R/W 都置位，则就有读写的权限)。然后针对当前处理的页目录项对应
// 的页表，设置需要复制的页面项数。如果是在内核空间，则仅需复制头 160 页对应的页表项
// (nr = 160)，对应于开始 640KB 物理内存。否则需要复制一个页表中的所有 1024 个页表项
// (nr = 1024)，可映射 4MB 物理内存。
171         *to_dir = ((unsigned long) to_page_table) | 7;
172         nr = (from==0)?0xA0:1024;
// 此时对于当前页表，开始循环复制指定的 nr 个内存页面表项。先取出源页表项内容，如果
// 当前源页面没有使用，则不用复制该表项，继续处理下一项。否则复位页表项中 R/W 标志
// (位 1 置 0)，即让页表项对应的内存页面只读。然后将该页表项复制到目的页表中。
173         for ( ; nr-- > 0 ; from_page_table++, to_page_table++) {
174             this_page = *from_page_table;
175             if (!(1 & this_page))
176                 continue;
177             this_page &= ~2;
178             *to_page_table = this_page;
// 如果该页表项所指物理页面的地址在 1MB 以上，则需要设置内存页面映射数组 mem_map[]，
// 于是计算页面号，并以它为索引在页面映射数组相应项中增加引用次数。而对于位于 1MB
// 以下的页面，说明是内核页面，因此不需要对 mem_map[] 进行设置。因为 mem_map[] 仅用
// 于管理主内存区中的页面使用情况。因此对于内核移动到任务 0 中并且调用 fork() 创建
// 任务 1 时 (用于运行 init())，由于此时复制的页面还仍然都在内核代码区域，因此以下
// 判断中的语句不会执行，任务 0 的页面仍然可以随时读写。只有当调用 fork() 的父进程
// 代码处于主内存区 (页面位置大于 1MB) 时才会执行。这种情况需要在进程调用 execve()，
// 并装载执行了新程序代码时才会出现。
// 180 行语句含义是令源页表项所指内存页也为只读。因为现在开始有两个进程共用内存区了。
// 若其中 1 个进程需要进行写操作，则可以通过页异常写保护处理为执行写操作的进程分配
// 1 页新空闲页面，也即进行写时复制 (copy on write) 操作。
179             if (this_page > LOW_MEM) {
180                 *from_page_table = this_page; // 令源页表项也只读。
181                 this_page -= LOW_MEM;
182                 this_page >>= 12;
183                 mem_map[this_page]++;
184             }
185         }
186     }
187     invalidate();          // 刷新页变换高速缓冲。
188     return 0;
189 }
190
191 /*
192  * This function puts a page in memory at the wanted address.
193  * It returns the physical address of the page gotten, 0 if
194  * out of memory (either when trying to access page-table or
195  * page.)
196  */
/*
* 下面函数将一内存页面放置 (映射) 到指定线性地址处。它返回页面
* 的物理地址，如果内存不够 (在访问页表或页面时)，则返回 0。

```



```

    */
    // 把一物理内存页面映射到线性地址空间指定处。
    // 或者说是把线性地址空间中指定地址 address 处的页面映射到主内存区页面 page 上。主要
    // 工作是在相关页目录项和页表项中设置指定页面的信息。若成功则返回物理页面地址。在
    // 处理缺页异常的 C 函数 do_no_page() 中会调用此函数。对于缺页引起的异常，由于任何缺
    // 页缘故而对页表作修改时，并不需要刷新 CPU 的页变换缓冲（或称 Translation Lookaside
    // Buffer - TLB），即使页表项中标志 P 被从 0 修改成 1。因为无效页项不会被缓冲，因此当
    // 修改了一个无效的页表项时不需要刷新。在此就表现为不用调用 Invalidate() 函数。
    // 参数 page 是分配的主内存区中某一页面（页帧，页框）的指针；address 是线性地址。
197 unsigned long put_page(unsigned long page, unsigned long address)
198 {
199     unsigned long tmp, *page_table;
200
201     /* NOTE !!! This uses the fact that _pg_dir=0 */
    /* 注意!!!这里使用了页目录基址_pg_dir=0 的条件 */
202
    // 首先判断参数给定物理内存页面 page 的有效性。如果该页面位置低于 LOW_MEM（1MB）或
    // 超出系统实际含有内存高端 HIGH_MEMORY，则发出警告。LOW_MEM 是主内存区可能有的最
    // 小起始位置。当系统物理内存小于或等于 6MB 时，主内存区起始于 LOW_MEM 处。再查看一
    // 下该 page 页面是否是已经申请的页面，即判断其在内存页面映射字节图 mem_map[] 中相
    // 应字节是否已经置位。若没有则需发出警告。
203     if (page < LOW_MEM || page >= HIGH_MEMORY)
204         printk("Trying to put page %p at %p\n", page, address);
205     if (mem_map[(page-LOW_MEM)>>12] != 1)
206         printk("mem_map disagrees with %p at %p\n", page, address);
    // 然后根据参数指定的线性地址 address 计算其在页目录表中对应的目录项指针，并从中取得
    // 二级页表地址。如果该目录项有效（P=1），即指定的页表在内存中，则从中取得指定页表
    // 地址放到 page_table 变量中。否则就申请一空闲页面给页表使用，并在对应目录项中置相
    // 应标志（7 - User、U/S、R/W）。然后将该页表地址放到 page_table 变量中。参见对 115
    // 行语句的说明。
207     page_table = (unsigned long *) ((address>>20) & 0xffc);
208     if ((*page_table)&1)
209         page_table = (unsigned long *) (0xfffff000 & *page_table);
210     else {
211         if (!(tmp=get_free_page()))
212             return 0;
213         *page_table = tmp|7;
214         page_table = (unsigned long *) tmp;
215     }
    // 最后在找到的页表 page_table 中设置相关页表项内容，即把物理页面 page 的地址填入表
    // 项同时置位 3 个标志（U/S、W/R、P）。该页表项在页表中的索引值等于线性地址位 21 --
    // 位 12 组成的 10 比特的值。每个页表共可有 1024 项（0 -- 0x3ff）。
216     page_table[(address>>12) & 0x3ff] = page | 7;
217     /* no need for invalidate */
    /* 不需要刷新页变换高速缓冲 */
218     return page;                // 返回物理页面地址。
219 }
220
    // 取消写保护页面函数。用于页异常中断过程中写保护异常的处理（写时复制）。
    // 在内核创建进程时，新进程与父进程被设置成共享代码和数据内存页面，并且所有这些页面
    // 均被设置成只读页面。而当新进程或原进程需要向内存页面写数据时，CPU 就会检测到这个
    // 情况并产生页面写保护异常。于是在这个函数中内核就会首先判断要写的页面是否被共享。
    // 若没有则把页面设置成可写然后退出。若页面是出于共享状态，则需要重新申请一新页面并

```

```

// 复制被写页面内容，以供写进程单独使用。共享被取消。本函数供下面 do_wp_page() 调用。
// 输入参数为页表项指针，是物理地址。[ un_wp_page -- Un-Write Protect Page]
221 void un_wp_page(unsigned long * table_entry)
222 {
223     unsigned long old_page, new_page;
224
// 首先取参数指定的页表项中物理页面位置（地址）并判断该页面是否是共享页面。如果原
// 页面地址大于内存低端 LOW_MEM（表示在主内存区中），并且其在页面映射字节图数组中
// 值为 1（表示页面仅被引用 1 次，页面没有被共享），则在该页面的页表项中置 R/W 标志
// （可写），并刷新页变换高速缓冲，然后返回。即如果该内存页面此时只被一个进程使用，
// 并且不是内核中的进程，就直接把属性改为可写即可，不用再重新申请一个新页面。
225     old_page = 0xfffff000 & *table_entry; // 取指定页表项中物理页面地址。
226     if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
227         *table_entry |= 2;
228         invalidate();
229         return;
230     }
// 否则就需要在主内存区内申请一页空闲页面给执行写操作的进程单独使用，取消页面共享。
// 如果原页面大于内存低端（则意味着 mem_map[] > 1，页面是共享的），则将原页面的页
// 面映射字节数组值递减 1。然后将指定页表项内容更新为新页面地址，并置可读写等标志
// （U/S、R/W、P）。在刷新页变换高速缓冲之后，最后将原页面内容复制到新页面上。
231     if (!(new_page=get_free_page()))
232         oom(); // Out of Memory。内存不够处理。
233     if (old_page >= LOW_MEM)
234         mem_map[MAP_NR(old_page)]--;
235     *table_entry = new_page | 7;
236     invalidate();
237     copy_page(old_page, new_page);
238 }
239
240 /*
241  * This routine handles present pages, when users try to write
242  * to a shared page. It is done by copying the page to a new address
243  * and decrementing the shared-page counter for the old page.
244  *
245  * If it's in code space we exit with a segment error.
246  */
/*
 * 当用户试图往一共享页面上写时，该函数处理已存在的内存页面（写时复制），
 * 它是通过将页面复制到一个新地址上并且递减原页面的共享计数值实现的。
 *
 * 如果它在代码空间，我们就显示段出错信息并退出。
 */
///// 执行写保护页面处理。
// 是写共享页面处理函数。是页异常中断处理过程中调用的 C 函数。在 page.s 程序中被调用。
// 参数 error_code 是进程在写写保护页面时由 CPU 自动产生，address 是页面线性地址。
// 写共享页面时，需复制页面（写时复制）。
247 void do_wp_page(unsigned long error_code, unsigned long address)
248 {
249     #if 0
250     /* we cannot do this yet: the estdio library writes to code space */
251     /* stupid, stupid. I really want the libc.a from GNU */
    /* 我们现在还不能这样做：因为 estdio 库会在代码空间执行写操作 */

```

```

/* 真是太愚蠢了。我真想从 GNU 得到 libc.a 库。*/
252     if (CODE_SPACE(address)) // 如果地址位于代码空间，则终止执行程序。
253         do_exit(SIGSEGV);
254 #endif
// 调用上面函数 un_wp_page() 来处理取消页面保护。但首先需要为其准备好参数。参数是
// 线性地址 address 指定页面在页表中的页表项指针，其计算方法是：
// ① ((address>>10) & 0xffc)：计算指定线性地址中页表项在页表中的偏移地址；因为
// 根据线性地址结构，(address>>12) 就是页表项中的索引，但每项占 4 个字节，因此乘
// 4 后：(address>>12)<<2 = (address>>10)&0xffc 就可得到页表项在表中的偏移地址。
// 与操作&0xffc 用于限制地址范围在一个页面内。又因为只移动了 10 位，因此最后 2 位
// 是线性地址低 12 位中的最高 2 位，也应屏蔽掉。因此求线性地址中页表项在页表中偏
// 移地址直观一些表示方法是(((address>>12) & 0x3ff)<<2)。
// ② (0xfffff000 & *((address>>20) & 0xffc))：用于取目录项中页表的地址值；其中，
// ((address>>20) & 0xffc)用于取线性地址中的目录索引项在目录表中的偏移位置。因为
// address>>22 是目录项索引值，但每项 4 个字节，因此乘以 4 后：(address>>22)<<2
// = (address>>20) 就是指定项在目录表中的偏移地址。&0xffc 用于屏蔽目录项索引值
// 中最后 2 位。因为只移动了 20 位，因此最后 2 位是页表索引的内容，应该屏蔽掉。而
// *((address>>20) & 0xffc) 则是取指定目录表项内容中对应页表的物理地址。最后与上
// 0xfffff000 用于屏蔽掉页目录项内容中的一些标志位（目录项低 12 位）。直观表示为
// (0xfffff000 & *((unsigned long *) ((address>>22) & 0x3ff)<<2)))。
// ③ 由①中页表项在页表中偏移地址加上 ②中目录表项内容中对应页表的物理地址即可
// 得到页表项的指针（物理地址）。这里对共享的页面进行复制。
255     un_wp_page((unsigned long *)
256                (((address>>10) & 0xffc) + (0xfffff000 &
257                *((unsigned long *) ((address>>20) & 0xffc)))));
258 }
259 }
260
//// 写页面验证。
// 若页面不可写，则复制页面。在 fork.c 中第 34 行被内存验证通用函数 verify_area() 调用。
// 参数 address 是指定页面的线性地址。
261 void write_verify(unsigned long address)
262 {
263     unsigned long page;
264
// 首先取指定线性地址对应的页目录项，根据目录项中的存在位（P）判断目录项对应的页表
// 是否存在（存在位 P=1?），若不存在（P=0）则返回。这样处理是因为对于不存在的页面没
// 有共享和写时复制可言，并且若程序对此不存在的页面执行写操作时，系统就会因为缺页异
// 常而去执行 do_no_page()，并为这个地方使用 put_page() 函数映射一个物理页面。
// 接着程序从目录项中取页表地址，加上指定页面在页表中的页表项偏移值，得对应地址的页
// 表项指针。在该表项中包含着给定线性地址对应的物理页面。
265     if (!(page = *((unsigned long *) ((address>>20) & 0xffc)) & 1))
266         return;
267     page &= 0xfffff000;
268     page += ((address>>10) & 0xffc);
// 然后判断该页表项中的位 1（R/W）、位 0（P）标志。如果该页面不可写（R/W=0）且存在，
// 那么就执行共享检验和复制页面操作（写时复制）。否则什么也不做，直接退出。
269     if ((3 & *((unsigned long *) page) == 1) /* non-writeable, present */
270         un_wp_page((unsigned long *) page);
271     return;
272 }
273
//// 取得一页空闲内存页并映射到指定线性地址处。

```

```

// get_free_page() 仅是申请取得了主内存区的一页物理内存。而本函数则不仅是获取到一页
// 物理内存页面，还进一步调用 put_page(), 将物理页面映射到指定的线性地址处。
// 参数 address 是指定页面的线性地址。
274 void get_empty_page(unsigned long address)
275 {
276     unsigned long tmp;
277
// 若不能取得一空闲页面，或者不能将所取页面放置到指定地址处，则显示内存不够的信息。
// 279 行上英文注释的含义是：free_page() 函数的参数 tmp 是 0 也没有关系，该函数会忽略
// 它并能正常返回。
278     if (!(tmp=get_free_page()) || !put_page(tmp, address)) {
279         free_page(tmp);          /* 0 is ok - ignored */
280         oom();
281     }
282 }
283
284 /*
285  * try_to_share() checks the page at address "address" in the task "p",
286  * to see if it exists, and if it is clean. If so, share it with the current
287  * task.
288  *
289  * NOTE! This assumes we have checked that p != current, and that they
290  * share the same executable.
291  */
/*
 * try_to_share() 在任务 "p" 中检查位于地址 "address" 处的页面，看页面是否存在，
 * 是否干净。如果是干净的话，就与当前任务共享。
 *
 * 注意！这里我们已假定 p != 当前任务，并且它们共享同一个执行程序。
 */
///// 尝试对当前进程指定地址处的页面进行共享处理。
// 当前进程与进程 p 是同一执行代码，也可以认为当前进程是由 p 进程执行 fork 操作产生的
// 进程，因此它们的代码内容一样。如果未对数据段内容作过修改那么数据段内容也应一样。
// 参数 address 是进程中的逻辑地址，即是当前进程欲与 p 进程共享页面的逻辑页面地址。
// 进程 p 是将被共享页面的进程。如果 p 进程 address 处的页面存在并且没有被修改过的话，
// 就让当前进程与 p 进程共享之。同时还需要验证指定的地址处是否已经申请了页面，若是
// 则出错，死机。返回：1 - 页面共享处理成功；0 - 失败。
292 static int try_to_share(unsigned long address, struct task_struct * p)
293 {
294     unsigned long from;
295     unsigned long to;
296     unsigned long from_page;
297     unsigned long to_page;
298     unsigned long phys_addr;
299
// 首先分别求得指定进程 p 中和当前进程中逻辑地址 address 对应的页目录项。为了计算方便
// 先求出指定逻辑地址 address 处的'逻辑'页目录项号，即以进程空间 (0 - 64MB) 算出的页
// 目录项号。该'逻辑'页目录项号加上进程 p 在 CPU 4G 线性空间中起始地址对应的页目录项，
// 即得到进程 p 中地址 address 处页面所对应的 4G 线性空间中的实际页目录项 from_page。
// 而'逻辑'页目录项号加上当前进程 CPU 4G 线性空间中起始地址对应的页目录项，即可最后
// 得到当前进程中地址 address 处页面所对应的 4G 线性空间中的实际页目录项 to_page。
300     from_page = to_page = ((address>>20) & 0xffc);
301     from_page += ((p->start_code>>20) & 0xffc);    // p 进程目录项。

```

```

302         to_page += ((current->start_code>>20) & 0xffc); // 当前进程目录项。

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

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

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

```



```

/* 对它们进行共享处理：写保护 */
327     *(unsigned long *) from_page &= ~2;
328     *(unsigned long *) to_page = *(unsigned long *) from_page;
// 随后刷新页变换高速缓冲。计算所操作物理页面的页面号，并将对应页面映射字节数组项中
// 的引用递增 1。最后返回 1，表示共享处理成功。
329     invalidate();
330     phys_addr -= LOW_MEM;
331     phys_addr >>= 12; // 得页面号。
332     mem_map[phys_addr]++;
333     return 1;
334 }
335
336 /*
337  * share_page() tries to find a process that could share a page with
338  * the current one. Address is the address of the wanted page relative
339  * to the current data space.
340  *
341  * We first check if it is at all feasible by checking executable->i_count.
342  * It should be >1 if there are other tasks sharing this inode.
343  */
/*
 * share_page() 试图找到一个进程，它可以与当前进程共享页面。参数 address 是
 * 当前进程数据空间中期望共享的某页面地址。
 *
 * 首先我们通过检测 executable->i_count 来查证是否可行。如果有其他任务已共享
 * 该 inode，则它应该大于 1。
 */
///// 共享页面处理。
// 在发生缺页异常时，首先看看能否与运行同一个执行文件的其他进程进行页面共享处理。
// 该函数首先判断系统中是否有另一个进程也在运行当前进程一样的执行文件。若有，则在
// 系统当前所有任务中寻找这样的任务。若找到了这样的任务就尝试与其共享指定地址处的
// 页面。若系统中没有其他任务正在运行与当前进程相同的执行文件，那么共享页面操作的
// 前提条件不存在，因此函数立刻退出。判断系统中是否有另一个进程也在执行同一个执行
// 文件的方法是利用进程任务数据结构中的 executable 字段。该字段指向进程正在执行程
// 序在内存中的 i 节点。根据该 i 节点的引用次数 i_count 我们可以进行这种判断。若
// executable->i_count 值大于 1，则表明系统中可能有两个进程在运行同一个执行文件，
// 于是可以再对任务结构数组中所有任务比较是否有相同的 executable 字段来最后确定多
// 个进程运行着相同执行文件的情况。
// 参数 address 是进程中的逻辑地址，即是当前进程欲与 p 进程共享页面的逻辑页面地址。
// 返回 1 - 共享操作成功，0 - 失败。
344 static int share_page(unsigned long address)
345 {
346     struct task_struct ** p;
347
// 首先检查一下当前进程的 executable 字段是否指向某执行文件的 i 节点，以判断本进程
// 是否有对应的执行文件。如果没有，则返回 0。如果 executable 的确指向某个 i 节点，
// 则检查该 i 节点引用计数值。如果当前进程运行的执行文件的内存 i 节点引用计数等于
// 1 (executable->i_count == 1)，表示当前系统中只有 1 个进程（即当前进程）在运行该
// 执行文件。因此无共享可言，直接退出函数。
348     if (!current->executable)
349         return 0;
350     if (current->executable->i_count < 2)
351         return 0;

```

```

// 否则搜索任务数组中所有任务。寻找与当前进程可共享页面的进程，即运行相同执行文件
// 的另一个进程，并尝试对指定地址的页面进行共享。如果找到某个进程 p，其 executable
// 字段值与当前进程的相同，则调用 try_to_share() 尝试页面共享。若共享操作成功，则
// 函数返回 1。否则返回 0，表示共享页面操作失败。
352     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
353         if (!*p)                                // 如果该任务项空闲，则继续寻找。
354             continue;
355         if (current == *p)                        // 如果就是当前任务，也继续寻找。
356             continue;
// 如果 executable 不等，表示运行的不是与当前进程相同的执行文件，因此也继续寻找。
357         if ((*p)->executable != current->executable)
358             continue;
359         if (try_to_share(address,*p))            // 尝试共享页面。
360             return 1;
361     }
362     return 0;
363 }
364
//// 执行缺页处理。
// 是访问不存在页面处理函数。页异常中断处理过程中调用的函数。在 page.s 程序中被调用。
// 函数参数 error_code 和 address 是进程在访问页面时由 CPU 因缺页产生异常而自动生成。
// error_code 指出出错类型，参见本章开始处的“内存页面出错异常”一节；address 是产生
// 异常的页面线性地址。
// 该函数首先尝试与已加载的相同文件进行页面共享，或者只是由于进程动态申请内存页面而
// 只需映射一页物理内存页即可。若共享操作不成功，那么只能从相应文件中读入所缺的数据
// 页面到指定线性地址处。
365 void do_no_page(unsigned long error_code,unsigned long address)
366 {
367     int nr[4];
368     unsigned long tmp;
369     unsigned long page;
370     int block,i;
371
// 首先取线性空间中指定地址 address 处页面地址。从而可算出指定线性地址在进程空间中
// 相对于进程基址的偏移长度值 tmp，即对应的逻辑地址。
372     address &= 0xfffff000;                        // address 处缺页页面地址。
373     tmp = address - current->start_code;          // 缺页页面对应逻辑地址。
// 若当前进程的 executable 节点指针空，或者指定地址超出（代码 + 数据）长度，则申请
// 一页物理内存，并映射到指定的线性地址处。executable 是进程正在运行的执行文件的 i
// 节点结构。由于任务 0 和任务 1 的代码在内核中，因此任务 0、任务 1 以及任务 1 派生的
// 没有调用过 execve() 的所有任务的 executable 都为 0。若该值为 0，或者参数指定的线性
// 地址超出代码加数据长度，则表明进程在申请新的内存页面存放堆或栈中数据。因此直接
// 调用取空闲页面函数 get_empty_page() 为进程申请一页物理内存并映射到指定线性地址
// 处。进程任务结构 字段 start_code 是线性地址空间中进程代码段地址，字段 end_data
// 是代码加数据长度。对于 Linux 0.11 内核，它的代码段和数据段起始基址相同。
374     if (!current->executable || tmp >= current->end_data) {
375         get_empty_page(address);
376         return;
377     }
// 否则说明所缺页面在进程执行影像文件范围内，于是就尝试共享页面操作，若成功则退出。
// 若不成功就只能申请一页物理内存页面 page，然后从设备上读取执行文件中的相应页面并
// 放置（映射）到进程页面逻辑地址 tmp 处。
378     if (share_page(tmp))                        // 尝试逻辑地址 tmp 处页面的共享。

```



```

379         return;
380         if (!(page = get_free_page()))           // 申请一页物理内存。
381             oom();
382 /* remember that 1 block is used for header */
383 /* 记住，（程序）头要使用 1 个数据块 */
384 // 因为块设备上存放的执行文件映像第 1 块数据是程序头结构，因此在读取该文件时需要跳过
385 // 第 1 块数据。所以需要首先计算缺页所在的数据块号。因为每块数据长度为 BLOCK_SIZE =
386 // 1KB，因此一页内存可存放 4 个数据块。进程逻辑地址 tmp 除以数据块大小再加 1 即可得出
387 // 缺少的页面在执行映像文件中的起始块号 block。根据这个块号和执行文件的 i 节点，我们
388 // 就可以从映射位图中找到对应块设备中对应的设备逻辑块号（保存在 nr[] 数组中）。利用
389 // bread_page() 即可把这 4 个逻辑块读入到物理页面 page 中。
390         block = 1 + tmp/BLOCK_SIZE;           // 执行文件中起始数据块号。
391         for (i=0 ; i<4 ; block++, i++)
392             nr[i] = bmap(current->executable, block); // 设备上对应的逻辑块号。
393         bread_page(page, current->executable->i_dev, nr); // 读设备上 4 个逻辑块。

394 // 在读设备逻辑块操作时，可能会出现这样一种情况，即在执行文件中的读取页面位置可能离
395 // 文件尾不到 1 个页面的长度。因此就可能读入一些无用的信息。下面的操作就是把这部分超
396 // 出执行文件 end_data 以后的部分清零处理。
397         i = tmp + 4096 - current->end_data;     // 超出的字节长度值。
398         tmp = page + 4096;                       // tmp 指向页面末端。
399         while (i-- > 0) {                         // 页面末端 i 字节清零。
400             tmp--;
401             *(char *)tmp = 0;
402         }
403 // 最后把引起缺页异常的一页物理页面映射到指定线性地址 address 处。若操作成功就返回。
404 // 否则就释放内存页，显示内存不够。
405         if (put_page(page, address))
406             return;
407         free_page(page);
408         oom();
409     }
410 }
411
412 //// 物理内存管理初始化。
413 // 该函数对 1MB 以上内存区域以页面为单位进行管理前的初始化设置工作。一个页面长度为
414 // 4KB 字节。该函数把 1MB 以上所有物理内存划分成一个个页面，并使用一个页面映射字节
415 // 数组 mem_map[] 来管理所有这些页面。对于具有 16MB 内存容量的机器，该数组共有 3840
416 // 项 ((16MB - 1MB)/4KB)，即可管理 3840 个物理页面。每当一个物理内存页面被占用时就
417 // 把 mem_map[] 中对应的的字节值增 1；若释放一个物理页面，就把对应字节值减 1。若字
418 // 节值为 0，则表示对应页面空闲；若字节值大于或等于 1，则表示对应页面被占用或被不
419 // 同程序共享占用。
420 // 在该版本的 Linux 内核中，最多能管理 16MB 的物理内存，大于 16MB 的内存将弃置不用。
421 // 对于具有 16MB 内存的 PC 机系统，在没有设置虚拟盘 RAMDISK 的情况下 start_mem 通常
422 // 是 4MB，end_mem 是 16MB。因此此时主内存区范围是 4MB—16MB，共有 3072 个物理页面可
423 // 供分配。而范围 0 - 1MB 内存空间用于内核系统（其实内核只使用 0 —640Kb，剩下的部
424 // 分被部分高速缓冲和设备内存占用）。
425 // 参数 start_mem 是用作页面分配的主内存区起始地址（已去除 RAMDISK 所占内存空间）。
426 // end_mem 是实际物理内存最大地址。而地址范围 start_mem 到 end_mem 是主内存区。
427 void mem_init(long start_mem, long end_mem)
428 {
429     int i;
430
431     // 首先将 1MB 到 16MB 范围内所有内存页面对应的内存映射字节数组项置为已占用状态，即各

```

```

// 项字节值全部设置成 USED (100)。PAGING_PAGES 被定义为(PAGING_MEMORY>>12)，即 1MB
// 以上所有物理内存分页后的内存页面数(15MB/4KB = 3840)。
403     HIGH_MEMORY = end_mem;                // 设置内存最高端 (16MB)。
404     for (i=0 ; i<PAGING_PAGES ; i++)
405         mem_map[i] = USED;
// 然后计算主内存区起始内存 start_mem 处页面对应内存映射字节数组中项号 i 和主内存区
// 页面数。此时 mem_map[] 数组的第 i 项正对应主内存区中第 1 个页面。最后将主内存区中
// 页面对应的数组项清零 (表示空闲)。对于具有 16MB 物理内存的系统，mem_map[] 中对应
// 4Mb--16Mb 主内存区的项被清零。
406     i = MAP_NR(start_mem);                // 主内存区起始位置处页面号。
407     end_mem -= start_mem;
408     end_mem >>= 12;                        // 主内存区中的总页面数。
409     while (end_mem-->0)
410         mem_map[i++] = 0;                // 主内存区页面对应字节值清零。
411 }
412
//// 计算内存空闲页面数并显示。
// [?? 内核中没有地方调用该函数，Linux 调试过程中用的 ]
413 void calc_mem(void)
414 {
415     int i, j, k, free=0;
416     long * pg_tbl;
417
// 扫描内存页面映射数组 mem_map[]，获取空闲页面数并显示。然后扫描所有页目录项 (除 0，
// 1 项)，如果页目录项有效，则统计对应页表中有效页面数，并显示。页目录项 0—3 被内核
// 使用，因此应该从第 5 个目录项 (i=4) 开始扫描。
418     for(i=0 ; i<PAGING_PAGES ; i++)
419         if (!mem_map[i]) free++;
420     printk("%d pages free (of %d)\n", free, PAGING_PAGES);
421     for(i=2 ; i<1024 ; i++) {
422         if (l&pg_dir[i]) {
423             pg_tbl=(long *) (0xffffffff & pg_dir[i]);
424             for(j=k=0 ; j<1024 ; j++)
425                 if (pg_tbl[j]&1)
426                     k++;
427             printk("Pg-dir[%d] uses %d pages\n", i, k);
428         }
429     }
430 }
431

```

13.4 page.s 程序

13.4.1 功能描述

该文件包括页异常中断处理程序 (中断 14)，主要分两种情况处理。一是由于缺页引起的页异常中断，通过调用 `do_no_page(error_code, address)` 来处理；二是由页写保护引起的页异常，此时调用页写保护处理函数 `do_wp_page(error_code, address)` 进行处理。其中的出错码(`error_code`)是由 CPU 自动产生并压入堆栈的，出现异常时访问的线性地址是从控制寄存器 CR2 中取得的。CR2 是专门用来存放页出错时的线性地址。

13.4.2 代码注释

程序 13-3 linux/mm/page.s

```

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

```

13.4.3 其他信息

13.4.3.1 页出错异常处理

当处理器在转换线性地址到物理地址的过程中检测到以下两种条件时，就会发生页异常中断，中断

14。

- o 当 CPU 发现对应页目录项或页表项的存在位（Present）标志为 0。
- o 当前进程没有访问指定页面的权限。

对于页异常处理中断，CPU 提供了两项信息用来诊断页异常和从中恢复运行。

- (1) 放在堆栈上的出错码。该出错码指出了异常是由于页不存在引起的还是违反了访问权限引起的；在发生异常时 CPU 的当前特权层；以及是读操作还是写操作。出错码的格式是一个 32 位的长字。但只用了最后的 3 个比特位。分别说明导致异常发生时的原因：
 - 位 2(U/S) - 0 表示在超级用户模式下执行，1 表示在用户模式下执行；
 - 位 1(W/R) - 0 表示读操作，1 表示写操作；
 - 位 0(P) - 0 表示页不存在，1 表示页级保护。
- (2) CR2(控制寄存器 2)。CPU 将造成异常的用于访问的线性地址存放在 CR2 中。异常处理程序可以使用这个地址来定位相应的页目录和页表项。如果在页异常处理程序执行期间允许发生另一个页异常，那么处理程序应该将 CR2 压入堆栈中。

第14章 头文件(include)

程序在使用一个函数之前，应该首先声明该函数。为了便于使用，通常的做法是把同一类函数或数据结构以及常数的声明放在一个头文件（header file）中。头文件中也可以包括任何相关的类型定义和宏（macros）。在程序源代码文件中则使用预处理指令“`#include`”来引用相关的头文件。

程序中如下形式的一条控制行语句将会使得该行被文件 *filename* 的内容替换掉：

```
# include <filename>
```

当然，文件名 *filename* 中不能包含 `>` 和换行字符以及 `"`、`'`、`\`、或 `/*` 字符。编译系统会在定义的一系列地方搜索这个文件。类似下面形式的控制行会让编译器首先在源程序所在目录中搜索 *filename* 文件：

```
# include "filename"
```

如果没有找到，编译器再执行同上面一样的搜索过程。在这种形式中，文件名 *filename* 中不能包含换行字符和 `"`、`'`、`\`、或 `/*` 字符，但允许使用 `>` 字符。

在一般应用程序源代码中，头文件与开发环境中的库文件有着不可分割的紧密联系，库中的每个函数都需要在头文件中加以声明。应用程序开发环境中的头文件（通常放置在系统/usr/include/目录中）可以看作是其所提供函数库（例如 libc.a）中函数的一个组成部分，是库函数的使用说明或接口声明。在编译器把源代码程序转换成目标模块后，链接程序（linker）会把程序所有的目标模块组合在一起，包括用到的任何库文件中的模块。从而构成一个可执行的程序。

对于标准 C 函数库来讲，其最基本的头文件有 15 个。每个头文件都表示出一类特定函数的功能说明或结构定义，例如 I/O 操作函数、字符处理函数等。有关标准函数库的详细说明及其实现可参照 Plauger 编著的《The Standard C Library》一书。

而对于本书所描述的内核源代码，其中涉及到的头文件则可以看作是对内核及其函数库所提供服务的概要说明，是内核及其相关程序专用的头文件。在这些头文件中主要描述了内核所用到的所有数据结构、初始化数据、常数和宏定义，也包括少量的程序代码。除了几个专用的头文件以外（例如块设备头文件 blk.h），Linux 0.11 内核中所用到的头文件都放在内核代码树的 include/目录中。因此编译 Linux 0.11 内核无需使用开发环境提供的位于/usr/include/目录下的任何头文件。当然，tools/build.c 程序除外。因为这个程序虽然被包含在内核源代码树中，但它只是一个用于组合创建内核映像文件的工具程序或应用程序，不会被链接到内核代码中。

从 0.95 版开始，内核代码树中的头文件需要复制到/usr/include/linux 目录下才能顺利地编译内核。即从该版内核开始头文件已经与开发环境使用的头文件合二为一。

14.1 include/目录下的文件

内核所用到的头文件都保存在 include/目录下。该目录下的文件见列表 11.1 所示。这里需要说明一点：为了方便使用和兼容性，Linux 在编制内核程序头文件时所使用的命名方式与标准 C 库头文件的命名方式相似，许多头文件的名称甚至其中的一些内容都与标准 C 库的头文件基本相同，但这些内核头文件仍然是内核源代码或与内核有紧密联系的程序专用的。在一个 Linux 系统中，它们与标准库的头文件并存。通常的做法是把这些头文件放置在标准库头文件目录中的子目录下，以让需要用到内核数据结构或常数的程序使用。

另外，也由于版权问题，Linux 试图重新编制一些头文件以取代具有版权限制的标准 C 库的头文件。因此这些内核源代码中的头文件与开发环境中的头文件有一些重叠的地方。在 Linux 系统中，列表 14-1

中的 `asm/`、`linux/` 和 `sys/` 三个子目录下的内核头文件通常需要复制到标准 C 库头文件所在的目录（`/usr/include`）中，而其他一些文件若与标准库的头文件没有冲突则可以直接放到标准库头文件目录下，或者改放到这里的三个子目录中。

`asm/` 目录下主要用于存放与计算机体系结构密切相关的函数声明或数据结构的头文件。例如 Intel CPU 端口 IO 汇编宏文件 `io.h`、中断描述符设置汇编宏头文件 `system.h` 等。`linux/` 目录下是 Linux 内核程序使用的一些头文件。其中包括调度程序使用的头文件 `sched.h`、内存管理头文件 `mm.h` 和终端管理数据结构文件 `tty.h` 等。而 `sys/` 目录下存放着几个与内核资源相关头文件。不过从 0.98 版开始，内核目录树下 `sys/` 目录中的头文件被全部移到了 `linux/` 目录下。

Linux 0.11 版内核中共有 32 个头文件（*.h），其中 `asm/` 子目录中含有 4 个，`linux/` 子目录中含有 10 个，`sys/` 子目录中含有 5 个。从下一节开始我们首先描述 `include/` 目录下的 13 个头文件，然后依次说明每个子目录中的文件。说明顺序按照文件名称排序进行。

列表 14-1 `linux/include/` 目录下的文件

名称	大小	最后修改时间 (GMT)	说明
 asm/		1991-09-17 13:08:31	
 linux/		1991-11-02 13:35:49	
 sys/		1991-09-17 15:06:07	
 a.out.h	6047 bytes	1991-09-17 15:10:49	m
 const.h	321 bytes	1991-09-17 15:12:39	m
 ctype.h	1049 bytes	1991-11-07 17:30:47	m
 errno.h	1268 bytes	1991-09-17 15:04:15	m
 fcntl.h	1374 bytes	1991-09-17 15:12:39	m
 signal.h	1762 bytes	1991-09-22 19:58:04	m
 stdarg.h	780 bytes	1991-09-17 15:02:23	m
 stddef.h	286 bytes	1991-09-17 15:02:17	m
 string.h	7881 bytes	1991-09-17 15:04:09	m
 termios.h	5325 bytes	1991-11-25 20:02:08	m
 time.h	734 bytes	1991-09-17 15:02:02	m
 unistd.h	6410 bytes	1991-11-25 20:18:55	m
 utime.h	225 bytes	1991-09-17 15:03:38	m

14.2 a.out.h 文件

14.2.1 功能描述

在 Linux 内核中，`a.out.h` 文件用于定义被加载的可执行文件结构。主要用于加载程序 `fs/exec.c` 中。该文件不属于标准 C 库，它是内核专用的头文件。但由于与标准库的头文件名没有冲突，因此在 Linux 系统中一般可以放置 `/usr/include/` 目录下，以供涉及相关内容的程序使用。该头文件中定义了目标文件的一种 `a.out`（Assembly out）格式。Linux 0.11 系统中使用的 `.o` 文件和可执行文件就采用了这种目标文件格式。

`a.out.h` 文件包括三个数据结构定义和一些相关的宏定义，因此文件可被相应地分成三个部分：

- 第 1—108 行给出并描述了目标文件执行头结构和相关的宏定义；
- 第 109—185 行对符号表项结构的定义和说明；
- 第 186—217 行对重定位表项结构进行定义和说明。

由于该文件内容比较多，因此对其中三个数据结构以及相关宏定义的详细说明放在程序列表后。

从 0.96 版内核开始，Linux 系统直接采用了 GNU 的同名头文件 a.out.h。因此造成在 Linux 0.9x 下编译的程序不能在 Linux 0.1x 系统上运行。下面对两个 a.out 头文件的不同之处进行分析，并说明如何让 0.9x 下编译的一些不是用动态链接库的执行文件也能在 0.1x 下运行。

Linux 0.11 使用的 a.out.h 文件与 GNU 同名文件的主要区别处在于 exec 结构的第一个字段 a_magic。GNU 的该文件字段名称是 a_info，并且把该字段又分成 3 个子域：标志域 (Flags)、机器类型域 (Machine Type) 和魔数域 (Magic Number)。同时为机器类型域定义了相应的宏 N_MACHTYPE 和 N_FLAGS。见图 14-1 所示。

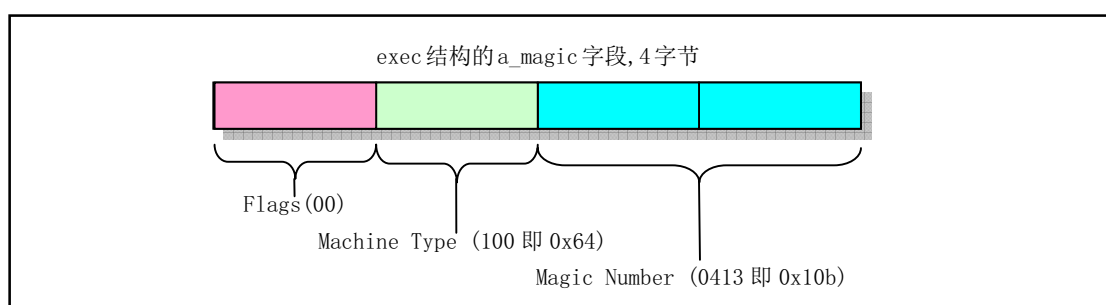


图 14-1 执行文件头结构 exec 中的第一个字段 a_magic (a_info)

在 Linux 0.9x 系统中，对于采用静态库连接的执行文件，图中各域注释中括号内的值是该字段的默认值。这种二进制执行文件开始处的 4 个字节是：

0x0b, 0x01, 0x64, 0x00

而这里的头文件仅定义了魔数域。因此，在 Linux 0.1x 系统中一个 a.out 格式的二进制执行文件开始的 4 个字节是：

0x0b, 0x01, 0x00, 0x00

可以看出，采用 GNU 的 a.out 格式的执行文件与 Linux 0.1x 系统上编译出的执行文件的区别仅在机器类型域。因此我们可以把 Linux 0.9x 上的 a.out 格式执行文件的机器类型域（第 3 个字节）清零，让其运行在 0.1x 系统中。只要被移植的执行文件所调用的系统调用都已经在 0.1x 系统中实现即可。在开始重新组建 Linux 0.1x 根文件系统中的很多命令时，作者就采用了这种方法。

在其他方面，GNU 的 a.out.h 头文件与这里的 a.out.h 没有什么区别。

14.2.2 代码注释

程序 14-1 linux/include/a.out.h

```
1 #ifndef A\_OUT\_H
2 #define A\_OUT\_H
```

```

3
4 #define GNU_EXEC_MACROS
5
// 第6--108行是该文件第1部分。定义目标文件执行结构以及相关操作的宏定义。
// 目标文件头结构。参见程序后的详细说明。
// =====
// unsigned long a_magic      // 执行文件魔数。使用 N_MAGIC 等宏访问。
// unsigned a_text           // 代码长度，字节数。
// unsigned a_data           // 数据长度，字节数。
// unsigned a_bss            // 文件中的未初始化数据区长度，字节数。
// unsigned a_syms           // 文件中的符号表长度，字节数。
// unsigned a_entry          // 执行开始地址。
// unsigned a_trsize         // 代码重定位信息长度，字节数。
// unsigned a_drsize         // 数据重定位信息长度，字节数。
// -----
6 struct exec {
7     unsigned long a_magic;    /* Use macros N_MAGIC, etc for access */
8     unsigned a_text;         /* length of text, in bytes */
9     unsigned a_data;         /* length of data, in bytes */
10    unsigned a_bss;           /* length of uninitialized data area for file, in bytes */
11    unsigned a_syms;          /* length of symbol table data in file, in bytes */
12    unsigned a_entry;         /* start address */
13    unsigned a_trsize;        /* length of relocation info for text, in bytes */
14    unsigned a_drsize;        /* length of relocation info for data, in bytes */
15 };
16
// 用于取上述 exec 结构中的魔数。
17 #ifndef N_MAGIC
18 #define N_MAGIC(exec) ((exec).a_magic)
19 #endif
20
21 #ifndef OMAGIC
22 /* Code indicating object file or impure executable. */
23 /* 指明为目标文件或者不纯的可执行文件的代号 */
24 // 历史上最早在 PDP-11 计算机上，魔数（幻数）是八进制数 0407（0x107）。它位于执行程序
25 // 头结构的开始处。原本是 PDP-11 的一条跳转指令，表示跳转到随后 7 个字后的代码开始处。
26 // 这样加载程序（loader）就可以在把执行文件放入内存后直接跳转到指令开始处运行。现在
27 // 已没有程序使用这种方法，但这个八进制数却作为识别文件类型的标志（魔数）保留了下来。
28 // OMAGIC 可以认为是 Old Magic 的意思。
29 #define OMAGIC 0407
30 /* Code indicating pure executable. */
31 /* 指明为纯可执行文件的代号 */ // New Magic, 1975 年以后开始使用。涉及虚存机制。
32 #define NMAGIC 0410 // 0410 == 0x108
33 /* Code indicating demand-paged executable. */
34 /* 指明为需求分页处理的可执行文件 */ // 其头结构占用文件开始处 1K 空间。
35 #define ZMAGIC 0413 // 0413 == 0x10b
36 #endif /* not OMAGIC */
37 // 另外还有一个 QMAGIC，是为了节约磁盘容量，把盘上执行文件的头结构与代码紧凑存放。
38 // 下面宏用于判断魔数字段的正确性。如果魔数不能被识别，则返回真。
39 #ifndef N_BADMAG
40 #define N_BADMAG(x) \
41     (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC \
42     && N_MAGIC(x) != ZMAGIC)

```

```

34 #endif
35
36 #define _N_BADMAG(x) \
37 (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC \
38 && N_MAGIC(x) != ZMAGIC)
39
40 // 目标文件头结构末端到 1024 字节之间的长度。
41 #define _N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct _exec))
42
43 // 下面宏用于操作目标文件的内容，包括.o 模块文件和可执行文件。
44
45 // 代码部分起始偏移值。
46 // 如果文件是 ZMAGIC 类型的，即是执行文件，那么代码部分是从执行文件的 1024 字节偏移处
47 // 开始；否则执行代码部分紧随执行头结构末端（32 字节）开始，即文件是模块文件（OMAGIC
48 // 类型）。
49 #ifndef N_TXTOFF
50 #define N_TXTOFF(x) \
51 (N_MAGIC(x) == ZMAGIC ? _N_HDROFF((x)) + sizeof (struct _exec) : sizeof (struct _exec))
52 #endif
53
54 // 数据部分起始偏移值。从代码部分末端开始。
55 #ifndef N_DATOFF
56 #define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
57 #endif
58
59 // 代码重定位信息偏移值。从数据部分末端开始。
60 #ifndef N_TRELOFF
61 #define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
62 #endif
63
64 // 数据重定位信息偏移值。从代码重定位信息末端开始。
65 #ifndef N_DRELOFF
66 #define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
67 #endif
68
69 // 符号表偏移值。从上面数据段重定位表末端开始。
70 #ifndef N_SYMOFF
71 #define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
72 #endif
73
74 // 字符串信息偏移值。在符号表之后。
75 #ifndef N_STROFF
76 #define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
77 #endif
78
79 // 下面对可执行文件被加载到内存（逻辑空间）中的位置情况进行操作。
80 /* Address of text segment in memory after it is loaded. */
81 /* 代码段加载后在内存中的地址 */
82 #ifndef N_TXTADDR
83 #define N_TXTADDR(x) 0 // 可见，代码段从地址 0 开始执行。
84 #endif
85
86 /* Address of data segment in memory after it is loaded.

```

```

73 Note that it is up to you to define SEGMENT_SIZE
74 on machines not listed here. */
/* 数据段加载后在内存中的地址。
   注意，对于下面没有列出名称的机器，需要你自己来定义
   对应的 SEGMENT_SIZE */
75 #if defined(vax) || defined(hp300) || defined(pyr)
76 #define SEGMENT_SIZE PAGE_SIZE
77 #endif
78 #ifdef hp300
79 #define PAGE_SIZE 4096
80 #endif
81 #ifdef sony
82 #define SEGMENT_SIZE 0x2000
83 #endif /* Sony. */
84 #ifdef is68k
85 #define SEGMENT_SIZE 0x20000
86 #endif
87 #if defined(m68k) && defined(PORTAR)
88 #define PAGE_SIZE 0x400
89 #define SEGMENT_SIZE PAGE_SIZE
90 #endif
91 // 这里，Linux 0.11 内核把内存页定义为 4KB，段大小定义为 1KB。因此没有使用上面的定义。
92 #define PAGE_SIZE 4096
93 #define SEGMENT_SIZE 1024
94 // 以段为界的大小（进位方式）。
95 #define N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))
96 // 代码段尾地址。
97 #define N_TXTENDADDR(x) (N_TXTADDR(x) + (x).a_text)
98 // 数据段开始地址。
99 // 如果文件是 OMAGIC 类型的，那么数据段就直接紧随代码段后面。否则的话数据段地址从代码
100 // 段后面段边界开始（1KB 边界对齐）。例如 ZMAGIC 类型的文件。
101 #ifndef N_DATADDR
102 #define N_DATADDR(x) \
103     (N_MAGIC(x) == OMAGIC ? (N_TXTENDADDR(x)) \
104     : (N_SEGMENT_ROUND (N_TXTENDADDR(x))))
105 #endif
106 /* Address of bss segment in memory after it is loaded. */
107 /* bss 段加载到内存以后的地址 */
108 /* 未初始化数据段 bbs 位于数据段后面，紧跟数据段。*/
109 #ifndef N_BSSADDR
110 #define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
111 #endif
112 // 第 110—185 行是第 2 部分。对目标文件中的符号表项和相关操作宏进行定义和说明。
113 // a.out 目标文件中符号表项结构（符号表记录结构）。参见程序后的详细说明。
114 #ifndef N_LIST_DECLARED
115 struct nlist {
116     union {

```

```

113     char *n_name;
114     struct nlist *n_next;
115     long n_strx;
116 } n_un;
117 unsigned char n_type;          // 该字节分成 3 个字段，146--154 行是相应字段的屏蔽码。
118 char n_other;
119 short n_desc;
120 unsigned long n_value;
121 };
122 #endif
123
124 // 下面定义 nlist 结构中 n_type 字段值的常量符号。
124 #ifndef N_UNDF
125 #define N_UNDF 0
126 #endif
127 #ifndef N_ABS
128 #define N_ABS 2
129 #endif
130 #ifndef N_TEXT
131 #define N_TEXT 4
132 #endif
133 #ifndef N_DATA
134 #define N_DATA 6
135 #endif
136 #ifndef N_BSS
137 #define N_BSS 8
138 #endif
139 #ifndef N_COMM
140 #define N_COMM 18
141 #endif
142 #ifndef N_FN
143 #define N_FN 15
144 #endif
145
146 // 以下 3 个常量定义是 nlist 结构中 n_type 字段的屏蔽码（八进程表示）。
146 #ifndef N_EXT
147 #define N_EXT 1                // 0x01 (0b0000, 0001) 符号是否是外部的（全局的）。
148 #endif
149 #ifndef N_TYPE
150 #define N_TYPE 036            // 0x1e (0b0001, 1110) 符号的类型位。
151 #endif
152 #ifndef N_STAB
153 #define N_STAB 0340           // STAB -- 符号表类型 (Symbol table types)。
154                                // 0xe0 (0b1110, 0000) 这几个比特用于符号调试器。
154 #endif
155
156 /* The following type indicates the definition of a symbol as being
157    an indirect reference to another symbol. The other symbol
158    appears as an undefined reference, immediately following this symbol.
159
160    Indirection is asymmetrical. The other symbol's value will be used
161    to satisfy requests for the indirect symbol, but not vice versa.
162    If the other symbol does not have a definition, libraries will
163    be searched to find a definition. */

```

```

/* 下面的类型指明对一个符号的定义是作为对另一个符号的间接引用。紧接该
 * 符号的其他的符号呈现为未定义的引用。
 *
 * 这种间接引用是不对称的。另一个符号的值将被用于满足间接符号的要求，
 * 但反之则不然。如果另一个符号没有定义，则将搜索库来寻找一个定义 */
164 #define N_INDR 0xa
165
166 /* The following symbols refer to set elements.
167    All the N_SET[ATDB] symbols with the same name form one set.
168    Space is allocated for the set in the text section, and each set
169    element's value is stored into one word of the space.
170    The first word of the space is the length of the set (number of elements).
171
172    The address of the set is made into an N_SETV symbol
173    whose name is the same as the name of the set.
174    This symbol acts like a N_DATA global symbol
175    in that it can satisfy undefined external references. */
/* 下面的符号与集合元素有关。所有具有相同名称 N_SET[ATDB] 的符号
   形成一个集合。在代码部分中已为集合分配了空间，并且每个集合元素
   的值存放在一个字（word）的空间中。空间的第一个字存有集合的长度（集合元素数目）。

   集合的地址被放入一个 N_SETV 符号中，它的名称与集合同名。
   在满足未定义的外部引用方面，该符号的行为象一个 N_DATA 全局符号。*/

176
177 /* These appear as input to LD, in a .o file. */
/* 以下这些符号在 .o 文件中是作为链接程序 LD 的输入。*/
178 #define N_SETA 0x14 /* Absolute set element symbol */ /* 绝对集合元素符号 */
179 #define N_SETT 0x16 /* Text set element symbol */ /* 代码集合元素符号 */
180 #define N_SETD 0x18 /* Data set element symbol */ /* 数据集合元素符号 */
181 #define N_SETB 0x1A /* Bss set element symbol */ /* Bss 集合元素符号 */
182
183 /* This is output from LD. */
/* 下面是 LD 的输出。*/
184 #define N_SETV 0x1C /* Pointer to set vector in data area. */
/* 指向数据区中集合向量。*/

185
186 #ifndef N_RELOCATION_INFO_DECLARED
187
188 /* This structure describes a single relocation to be performed.
189    The text-relocation section of the file is a vector of these structures,
190    all of which apply to the text section.
191    Likewise, the data-relocation section applies to the data section. */
/* 下面结构描述单个重定位操作的执行。
   文件的代码重定位部分是这些结构的一个数组，所有这些适用于代码部分。
   类似地，数据重定位部分用于数据部分。*/

192 // a.out 目标文件中代码和数据重定位信息结构。
193 struct relocation_info
194 {
195     /* Address (within segment) to be relocated. */
    /* 段内需要重定位的地址。*/
196     int r_address;
197     /* The meaning of r_symbolnum depends on r_extern. */

```

```

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

```

14.2.3 其他信息

14.2.3.1 a.out 执行文件格式

Linux 内核 0.11 版仅支持 a.out(Assembly out)执行文件和目标文件的格式，虽然这种格式目前已经渐渐不用，而使用功能更为齐全的 ELF (Executable and Link Format) 格式，但是由于其简单性，作为学习入门的材料正好比较适用。下面全面介绍一下 a.out 格式。

在头文件 a.out.h 中声明了三个数据结构以及一些宏。这些数据结构描述了系统上目标文件的结构。在 Linux 0.11 系统中，编译产生的目标模块文件（简称模块文件）和链接生成的二进制可执行文件均采用 a.out 格式。这里统称为目标文件。一个目标文件共可有 7 部分（七节）组成。它们依次为：

- a) **执行头部分** (exec header)。执行文件头部分。该部分中含有一些参数 (exec 结构)，内核使用这些参数把执行文件加载到内存中并执行，而链接程序(ld)使用这些参数将一些模块文件组合成一个可执行文件。这是目标文件唯一必要的组成部分。
- b) **代码段部分** (text segment)。含有程序执行时被加载到内存中的指令代码和相关数据。可以以只读形式被加载。

- c) **数据段部分 (data segment)**。这部分含有已经初始化过的数据，总是被加载到可读写的内存中。
- d) **代码重定位部分 (text relocations)**。这部分含有供链接程序使用的记录数据。在组合目标模块文件时用于定位代码段中的指针或地址。
- e) **数据重定位部分 (data relocations)**。类似于代码重定位部分的作用，但是用于数据段中指针的重定位。
- f) **符号表部分 (symbol table)**。这部分同样含有供链接程序使用的记录数据，用于在二进制目标模块文件之间对命名的变量和函数（符号）进行交叉引用。
- g) **字符串表部分 (string table)**。该部分含有与符号名相对应的字符串。

每个目标文件均以执行数据结构 (exec structure) 开始。该数据结构的形式如下：

```
struct exec {
    unsigned long a_magic      // 目标文件魔数。使用 N_MAGIC 等宏访问。
    unsigned a_text           // 代码长度，字节数。
    unsigned a_data           // 数据长度，字节数。
    unsigned a_bss            // 文件中的未初始化数据区长度，字节数。
    unsigned a_syms           // 文件中的符号表长度，字节数。
    unsigned a_entry          // 执行开始地址。
    unsigned a_trsize         // 代码重定位信息长度，字节数。
    unsigned a_drsize         // 数据重定位信息长度，字节数。
};
```

各个字段的功能如下：

- **a_magic** 该字段含有三个子字段，分别是标志字段、机器类型标识字段和魔数字段，参见图 11-1 所示。不过对于 Linux 0.11 系统其目标文件只使用了其中的魔数字子字段，并使用宏 N_MAGIC() 来访问，它唯一地确定了二进制执行文件与其他加载的文件之间的区别。该子字段中必须包含以下值之一：
 - ◆ **OMAGIC** 表示代码和数据段紧随在执行头后面并且是连续存放的。内核将代码和数据段都加载到可读写内存中。编译器编译出的目标文件的魔数是 OMAGIC（八进制 0407）。
 - ◆ **NMAGIC** 同 OMAGIC 一样，代码和数据段紧随在执行头后面并且是连续存放的。然而内核将代码加载到了只读内存中，并把数据段加载到代码段后下一页可读写内存边界开始。
 - ◆ **ZMAGIC** 内核在必要时从二进制执行文件中加载独立的页面。执行头部、代码段和数据段都被链接程序处理成多个页面大小的块。内核加载的代码页面是只读的，而数据段的页面是可写的。链接生成的可执行文件的魔数即是 ZMAGIC（0413，即 0x10b）。
- **a_text** 该字段含有代码段的长度值，字节数。
- **a_data** 该字段含有数据段的长度值，字节数。
- **a_bss** 含有 ‘bss 段’ 的长度，内核用其设置在数据段后初始的 break (brk)。内核在加载程序时，这段可写内存显现出处于数据段后面，并且初始时为全零。
- **a_syms** 含有符号表部分的字节长度值。
- **a_entry** 含有内核将执行文件加载到内存中以后，程序执行起始点的内存地址。
- **a_trsize** 该字段含有代码重定位表的大小，是字节数。
- **a_drsize** 该字段含有数据重定位表的大小，是字节数。

在 a.out.h 头文件中定义了几个宏，这些宏使用 exec 结构来测试一致性或者定位执行文件中各个部分（节）的位置偏移值。这些宏有：

N_BADMAG(exec) 如果 a_magic 字段不能被识别，则返回非零值。

N_TXTOFF(exec) 代码段的起始位置字节偏移值。
 N_DATOFF(exec) 数据段的起始位置字节偏移值。
 N_DRELOFF(exec) 数据重定位表的起始位置字节偏移值。
 N_TRELOFF(exec) 代码重定位表的起始位置字节偏移值。
 N_SYMOFF(exec) 符号表的起始位置字节偏移值。
 N_STROFF(exec) 字符串表的起始位置字节偏移值。

重定位记录具有标准的格式，它使用重定位信息(relocation_info)结构来描述，如下所示。

```
struct relocation_info
{
    int r_address;           // 段内需要重定位的地址。
    unsigned int r_symbolnum:24; // 含义与 r_extern 有关。指定符号表中一个符号或者一个段。
    unsigned int r_pcrel:1;    // PC 相关标志。
    unsigned int r_length:2;   // 要被重定位字段长度（2 的次方）。若值是 2 则 1<<2 字节数。
    unsigned int r_extern:1;   // 1 => 以符号的值重定位。 0 => 以段的地址进行重定位。
    unsigned int r_pad:4;      // 没有使用的 4 个比特位，但最好将它们复位掉。
};
```

该结构中各字段的含义如下：

- **r_address** 该字段含有需要链接程序处理（编辑）的指针的字节偏移值。代码重定位的偏移值是从代码段开始处计数的，数据重定位的偏移值是从数据段开始处计算的。链接程序会将已经存储在该偏移处的值与使用重定位记录计算出的新值相加。
- **r_symbolnum** 该字段含有符号表中一个符号结构的序号值（不是字节偏移值）。链接程序在算出符号的绝对地址以后，就将该地址加到正在进行重定位的指针上。（如果 **r_extern** 比特位是 0，那么情况就不同，见下面。）
- **r_pcrel** 如果设置了该位，链接程序就认为正在更新一个指针，该指针使用 **pc** 相关寻址方式，是属于机器码指令部分。当运行程序使用这个被重定位的指针时，该指针的地址被隐式地加到该指针上。
- **r_length** 该字段含有指针长度的 2 的次方值：0 表示 1 字节长，1 表示 2 字节长，2 表示 4 字节长。
- **r_extern** 如果被置位，表示该重定位需要一个外部引用；此时链接程序必须使用一个符号地址来更新相应指针。当该位是 0 时，则重定位是“局部”的；链接程序更新指针以反映各个段加载地址中的变化，而不是反映一个符号值的变化。在这种情况下，**r_symbolnum** 字段的内容是一个 **n_type** 值；这类字段告诉链接程序被重定位的指针指向那个段。
- **r_pad** Linux 系统中没有使用的 4 个比特位。在写一个目标文件时最好全置 0。

符号将名称映射为地址（或者更通俗地讲是字符串映射到值）。由于链接程序对地址的调整，一个符号的名称必须用来表示其地址，直到已被赋予一个绝对地址值。符号是由符号表中固定长度的记录以及字符串表中的可变长度名称组成。符号表是 **nlist** 结构的一个数组，如下所示。

```
struct nlist {
    union {
        char      *n_name;
        struct nlist *n_next;
        long       n_strx;
    };
};
```

```

} n_un;
unsigned char n_type;           // 该字节分成 3 个字段，146-154 行是相应字段的屏蔽码。
char          n_other;
short         n_desc;
unsigned long n_value;
};

```

其中各字段的含义为：

- **n_un.n_strx** 含有本符号的名称在字符串表中的字节偏移值。当程序使用 `nlist()` 函数访问一个符号表时，该字段被替换为 `n_un.n_name` 字段，这是内存中字符串的指针。
- **n_type** 用于链接程序确定如何更新符号的值。使用第 146--154 行开始的位屏蔽(bitmasks) 码可以将 8 比特宽度的 `n_type` 字段分割成三个子字段，见图 14-2 所示。对于 `N_EXT` 类型位置的符号，链接程序将它们看作是“外部的”符号，并且允许其他二进制目标文件对它们的引用。`N_TYPE` 屏蔽码用于链接程序感兴趣的比特位：
 - ◆ `N_UNDF` 一个未定义的符号。链接程序必须在其他二进制目标文件中定位一个具有相同名称的外部符号，以确定该符号的绝对数据值。特殊情况下，如果 `n_type` 字段是非零值，并且没有二进制文件定义了这个符号，则链接程序在 `BSS` 段中将该符号解析为一个地址，保留长度等于 `n_value` 的字节。如果符号在多于一个二进制目标文件中都没有定义并且这些二进制目标文件对其长度值设置均不一致，则链接程序将选择所有二进制目标文件中最大的长度。
 - ◆ `N_ABS` 一个绝对符号。链接程序不会更新一个绝对符号。
 - ◆ `N_TEXT` 一个代码符号。该符号的值是代码地址，链接程序在合并二进制目标文件时会更新其值。
 - ◆ `N_DATA` 一个数据符号；与 `N_TEXT` 类似，但是用于数据地址。对应代码和数据符号的值不是文件的偏移值而是地址；为了找出文件的偏移，就有必要确定相关部分开始加载的地址并减去它，然后加上该部分的偏移。
 - ◆ `N_BSS` 一个 `BSS` 符号；与代码或数据符号类似，但在二进制目标文件中没有对应的偏移。
 - ◆ `N_FN` 一个文件名符号。在合并二进制目标文件时，链接程序会将该符号插入在二进制文件中的符号之前。符号的名称就是给予链接程序的文件名，而其值是二进制文件中首个代码段地址。链接和加载时不需要文件名符号，但对于调式程序非常有用。
 - ◆ `N_STAB` 屏蔽码用于选择符号调式程序(例如 `gdb`)感兴趣的位；其值在 `stab()` 中说明。
- **n_other** 该字段按照 `n_type` 确定的段，提供有关符号重定位操作的符号独立性信息。目前，`n_other` 字段的最低 4 位含有两个值之一：`AUX_FUNC` 和 `AUX_OBJECT`（有关定义参见 `<link.h>`）。`AUX_FUNC` 将符号与可调用的函数相关，`AUX_OBJECT` 将符号与数据相关，而不管它们是位于代码段还是数据段。该字段主要用于链接程序 `ld`，用于动态可执行程序的创建。
- **n_desc** 保留给调式程序使用；链接程序不对其进行处理。不同的调试程序将该字段用作不同的用途。
- **n_value** 含有符号的值。对于代码、数据和 `BSS` 符号，这是一个地址；对于其他符号（例如调式程序符号），值可以是任意的。

字符串表是由长度为 `unsigned long` 后跟一 `null` 结尾的符号字符串组成。长度代表整个表的字节大小，所以在 32 位的机器上其最小值（即第 1 个字符串的偏移）总是 4。

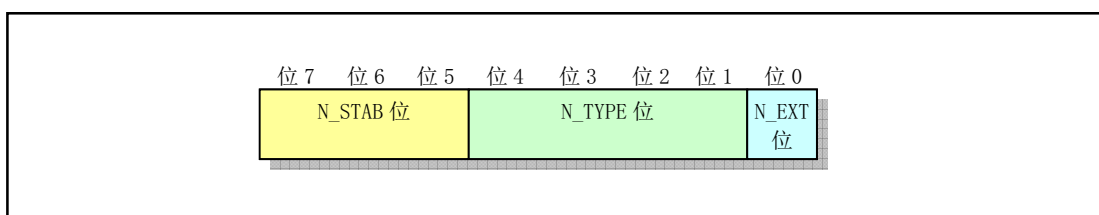


图 14-2 符号类型属性 n_type 字段

14.3 const.h 文件

14.3.1 功能描述

该文件定义了 i 节点中文件属性和类型 i_mode 字段所用的一些标志位常量符号。

14.3.2 代码注释

程序 14-2 linux/include/const.h

```

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

```

14.4 ctype.h 文件

14.4.1 功能描述

该文件是关于字符测试和处理的头文件，也是标准 C 库的头文件之一。其中定义了一些有关字符类型判断和转换的宏。例如判断一个字符 c 是一个数字字符（isdigit(c)）还是一个空格（isspace(c)）。在处理过程中使用了一个数组或表（定义在 lib/ctype.c 中），该数组定义了 ASCII 码表中所有字符的属性和类型。当使用宏时，字符代码是作为表_ctype[] 中的索引值，从表中获取一个字节，于是可得到相关的比特位。

另外，以两个下划线开头或者以一个下划线再加一个大写字母开头的宏名称通常都保留给头文件编制者使用。例如名称 `__abc` 和 `_SP`。

14.4.2 代码注释

程序 14-3 linux/include/ctype.h

```

1 #ifndef CTYPE_H
2 #define CTYPE_H
3
4 #define U      0x01    /* upper */           // 该比特位用于大写字母[A-Z]。
5 #define L      0x02    /* lower */           // 该比特位用于小写字母[a-z]。
6 #define D      0x04    /* digit */           // 该比特位用于数字[0-9]。
7 #define C      0x08    /* cntrl */           // 该比特位用于控制字符。
8 #define P      0x10    /* punct */           // 该比特位用于标点字符。
9 #define S      0x20    /* white space (space/lf/tab) */ // 空白字符，如空格、\t、\n等。
10 #define X      0x40    /* hex digit */        // 该比特位用于十六进制数字。
11 #define SP     0x80    /* hard space (0x20) */   // 该比特位用于空格字符(0x20)。
12
13 extern unsigned char ctype[];    // 字符特性数组(表)，定义各个字符对应上面的属性。
14 extern char ctmp;                // 一个临时字符变量(在定义lib/ctype.c中)。
15
16 // 下面是一些确定字符类型的宏。
17 #define isalnum(c) ((ctype+1)[c]&(U|L|D))    // 是字符或数字[A-Z]、[a-z]或[0-9]。
18 #define isalpha(c) ((ctype+1)[c]&(U|L))        // 是字符。
19 #define iscntrl(c) ((ctype+1)[c]&(C))            // 是控制字符。
20 #define isdigit(c) ((ctype+1)[c]&(D))            // 是数字。
21 #define isgraph(c) ((ctype+1)[c]&(P|U|L|D))    // 是图形字符。
22 #define islower(c) ((ctype+1)[c]&(L))            // 是小写字母。
23 #define isprint(c) ((ctype+1)[c]&(P|U|L|D|SP)) // 是可打印字符。
24 #define ispunct(c) ((ctype+1)[c]&(P))            // 是标点符号。
25 #define isspace(c) ((ctype+1)[c]&(S))            // 是空白字符如空格、\f、\n、\r、\t、\v。
26 #define isupper(c) ((ctype+1)[c]&(U))            // 是大写字母。
27 #define isxdigit(c) ((ctype+1)[c]&(D|X))        // 是十六进制数字。
28
29 // 在下面两个定义中，宏参数前使用了前缀(unsigned)，因此c应该加括号，即表示成(c)。
30 // 因为在程序中c可能是一个复杂的表达式。例如，如果参数是a + b，若不加括号，则在宏定
31 // 义中变成了：(unsigned) a + b。这显然不对。加了括号就能正确表示成(unsigned) (a + b)。
32 #define isascii(c) (((unsigned) c)<=0x7f)          // 是ASCII字符。
33 #define toascii(c) (((unsigned) c)&0x7f)           // 转换成ASCII字符。
34
35 // 以下两个宏定义中使用一个临时变量_ctmp的原因是：在宏定义中，宏的参数只能被使用一次。
36 // 但对于多线程来说这是不安全的，因为两个或多个线程可能在同一时刻使用这个公共临时变量。
37 // 因此从Linux 2.2.x版本开始更改为使用两个函数来取代这两个宏定义。
38 #define tolower(c) (ctmp=c, isupper(ctmp)? ctmp-'A'+'a': ctmp) // 转换成小写字母。
39 #define toupper(c) (ctmp=c, islower(ctmp)? ctmp-'a'+'A': ctmp) // 转换成大写字母。
40
41 #endif

```

14.5 errno.h 文件

14.5.1 功能描述

在系统或者标准 C 语言中有个名为 `errno` 的变量，关于在 C 标准中是否需要这个变量，在 C 标准化组织（X3J11）中引起了很大争论。但是争论的结果是没有去掉 `errno`，反而创建了名称为“`errno.h`”的头文件。因为标准化组织希望每个库函数或数据对象都需要在一个相应的标准头文件中作出声明。

主要原因在于：对于内核中的每个系统调用，如果其返回值就是指定系统调用的结果值的话，就很难报告出错情况。如果让每个函数返回一个对/错指示值，而结果值另行返回，就不能很方便地得到系统调用的结果值。解决的办法之一是将这两种方式加以组合：对于一个特定的系统调用，可以指定一个与有效结果值范围有区别的出错返回值。例如对于指针可以采用 `null` 值，对于 `pid` 可以返回 -1 值。在许多其他情况下，只要不与结果值冲突都可以采用 -1 来表示出错值。但是标准 C 库函数返回值仅告知是否发生出错，还必须从其他地方了解出错的类型，因此采用了 `errno` 这个变量。为了与标准 C 库的设计机制兼容，Linux 内核中的库文件也采用了这种处理方法。因此也借用了标准 C 的这个头文件。相关例子可参见 `lib/open.c` 程序以及 `unistd.h` 中的系统调用宏定义。在某些情况下，程序虽然从返回的 -1 值知道出错了，但想知道具体的出错号，就可以通过读取 `errno` 的值来确定最后一次错误的出错号。

本文件虽然只是定义了 Linux 系统中的一些出错码（出错号）的常量符号，而且 Linus 考虑程序的兼容性也想把这些符号定义成与 POSIX 标准中的一样。但是不要小看这个简单的代码，该文件也是 SCO 公司指责 Linux 操作系统侵犯其版权所列出的文件的之一。为了研究这个侵权问题，在 2003 年 12 月份，10 多个当前 Linux 内核的顶级开发人员在网上商讨对策。其中包括 Linus、Alan Cox、H.J.Lu、Mitchell Blank Jr. 由于当前内核版本（2.4.x）中的 `errno.h` 文件从 0.96c 版内核开始就没有变化过，他们就一直“跟踪”到这些老版本的内核代码中。最后 Linus 发现该文件是从 H.J.Lu 当时维护的 Libc 2.x 库中利用程序自动生成的，其中包括了一些与 SCO 拥有版权的 UNIX 老版本（V6、V7 等）相同的变量名。

14.5.2 代码注释

程序 14-4 linux/include/errno.h

```

1 #ifndef ERRNO\_H
2 #define ERRNO\_H
3
4 /*
5  * ok, as I hadn't got any other source of information about
6  * possible error numbers, I was forced to use the same numbers
7  * as minix.
8  * Hopefully these are posix or something. I wouldn't know (and posix
9  * isn't telling me - they want $$$ for their f***ing standard).
10  *
11  * We don't use the _SIGN cludge of minix, so kernel returns must
12  * see to the sign by themselves.
13  *
14  * NOTE! Remember to change strerror() if you change this file!
15  */
/*
 * ok, 由于我没有得到任何其他有关出错号的资料，我只能使用与 minix 系统
 * 相同的出错号了。
 * 希望这些是 POSIX 兼容的或者在一定程度上是这样的，我不知道（而且 POSIX

```

```

* 没有告诉我 - 要获得他们的混蛋标准需要出钱)。
*
* 我们没有使用 minix 那样的_SIGN 簇，所以内核的返回值必须自己辨别正负号。
*
* 注意！如果你改变该文件的话，记着也要修改 strerror() 函数。
*/

16 // 系统调用以及很多库函数返回一个特殊的值以表示操作失败或出错。这个值通常选择-1 或者
// 其他一些特定的值来表示。但是这个返回值仅说明错误发生了。 如果需要知道出错的类型，
// 就需要查看表示系统出错号的变量 errno。该变量即在 errno.h 文件中声明。在程序开始执
// 行时该变量值被初始化为 0。
17 extern int errno;
18
// 在出错时，系统调用会把出错号放在变量 errno 中（负值），然后返回-1。因此程序若需要知
// 道具体错误号，就需要查看 errno 的值。
19 #define ERROR 99 // 一般错误。
20 #define EPERM 1 // 操作没有许可。
21 #define ENOENT 2 // 文件或目录不存在。
22 #define ESRCH 3 // 指定的进程不存在。
23 #define EINTR 4 // 中断的系统调用。
24 #define EIO 5 // 输入/输出错。
25 #define ENXIO 6 // 指定设备或地址不存在。
26 #define E2BIG 7 // 参数列表太长。
27 #define ENOEXEC 8 // 执行程序格式错误。
28 #define EBADF 9 // 文件句柄(描述符)错误。
29 #define ECHILD 10 // 子进程不存在。
30 #define EAGAIN 11 // 资源暂时不可用。
31 #define ENOMEM 12 // 内存不足。
32 #define EACCES 13 // 没有许可权限。
33 #define EFAULT 14 // 地址错。
34 #define ENOTBLK 15 // 不是块设备文件。
35 #define EBUSY 16 // 资源正忙。
36 #define EEXIST 17 // 文件已存在。
37 #define EXDEV 18 // 非法连接。
38 #define ENODEV 19 // 设备不存在。
39 #define ENOTDIR 20 // 不是目录文件。
40 #define EISDIR 21 // 是目录文件。
41 #define EINVAL 22 // 参数无效。
42 #define ENFILE 23 // 系统打开文件数太多。
43 #define EMFILE 24 // 打开文件数太多。
44 #define ENOTTY 25 // 不恰当的 IO 控制操作(没有 tty 终端)。
45 #define ETXTBSY 26 // 不再使用。
46 #define EFBIG 27 // 文件太大。
47 #define ENOSPC 28 // 设备已满（设备已经没有空间）。
48 #define ESPIPE 29 // 无效的文件指针重定位。
49 #define EROFS 30 // 文件系统只读。
50 #define EMLINK 31 // 连接太多。
51 #define EPIPE 32 // 管道错。
52 #define EDOM 33 // 域(domain)出错。
53 #define ERANGE 34 // 结果太大。
54 #define EDEADLK 35 // 避免资源死锁。
55 #define ENAMETOOLONG 36 // 文件名太长。
56 #define ENOLCK 37 // 没有锁定可用。

```

```

57 #define ENOSYS          38          // 功能还没有实现。
58 #define ENOTEMPTY      39          // 目录不空。
59
60 #endif
61

```

14.6 fcntl.h 文件

14.6.1 功能描述

文件控制选项头文件。主要定义了文件控制函数 `fcntl()` 和文件创建或打开函数中用到的一些选项。`fcntl()` 函数在 `linux/fs/fcntl.c` 文件第 47 行开始的代码中实现，被用于对文件描述符（句柄）执行各种指定的操作，具体的操作由函数参数 `cmd`（命令）指定。

14.6.2 代码注释

程序 14-5 linux/include/fcntl.h

```

1 #ifndef FCNTL_H
2 #define FCNTL_H
3
4 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
5
6 /* open/fcntl - NOCTTY, NDELAY isn't implemented yet */
   /* open/fcntl - NOCTTY 和 NDELAY 现在还没有实现 */
7 #define O_ACCMODE          0003          // 文件访问模式屏蔽码。
   // 打开文件 open() 和文件控制函数 fcntl() 使用的文件访问模式。同时只能使用三者之一。
8 #define O_RDONLY          00          // 以只读方式打开文件。
9 #define O_WRONLY          01          // 以只写方式打开文件。
10 #define O_RDWR           02          // 以读写方式打开文件。
   // 下面是文件创建和操作标志，用于 open()。可与上面访问模式用 '位或' 的方式一起使用。
11 #define O_CREAT           00100      /* not fcntl */ // 如果文件不存在就创建。fcntl 函数不用。
12 #define O_EXCL            00200      /* not fcntl */ // 独占使用文件标志。
13 #define O_NOCTTY          00400      /* not fcntl */ // 不分配控制终端。
14 #define O_TRUNC           01000      /* not fcntl */ // 若文件已存在且是写操作，则长度截为 0。
15 #define O_APPEND          02000      // 以添加方式打开，文件指针置为文件尾。
16 #define O_NONBLOCK        04000      /* not fcntl */ // 非阻塞方式打开和操作文件。
17 #define O_NDELAY          O_NONBLOCK // 非阻塞方式打开和操作文件。
18
19 /* Defines for fcntl-commands. Note that currently
20  * locking isn't supported, and other things aren't really
21  * tested.
22  */
   /* 下面定义了 fcntl 的命令。注意目前锁定命令还没有支持，而其他
   * 命令实际上还没有测试过。
   */
   // 文件句柄(描述符)操作函数 fcntl() 的命令 (cmd)。
23 #define F_DUPFD           0          /* dup */ // 拷贝文件句柄为最小数值的句柄。
24 #define F_GETFD           1          /* get f_flags */ // 取句柄标志。仅 1 个标志 FD_CLOEXEC。

```

```

25 #define F_SETFD          2      /* set f_flags */ // 设置文件句柄标志。
26 #define F_GETFL          3      /* more flags (cloexec) */ // 取文件状态标志和访问模式。
27 #define F_SETFL          4      // 设置文件状态标志和访问模式。
    // 下面是文件锁定命令。fcntl() 的第三个参数 lock 是指向 flock 结构的指针。
28 #define F_GETLK          5      /* not implemented */ // 返回阻止锁定的 flock 结构。
29 #define F_SETLK          6      // 设置(F_RDLCK 或 F_WRLCK)或清除(F_UNLCK)锁定。
30 #define F_SETLKW         7      // 等待设置或清除锁定。
31
32 /* for F_[GET/SET]FL */
    /* 用于 F_GETFL 或 F_SETFL */
    // 在执行 exec() 簇函数时需要关闭的文件句柄。(执行时关闭 - Close On EXECution)
33 #define FD_CLOEXEC       1      /* actually anything with low bit set goes */
    /* 实际上只要低位为 1 即可 */
34
35 /* Ok, these are locking features, and aren't implemented at any
36  * level. POSIX wants them.
37  */
    /* OK, 以下是锁定类型, 任何函数中都还没有实现。POSIX 标准要求这些类型。
    */
38 #define F_RDLCK          0      // 共享或读文件锁定。
39 #define F_WRLCK          1      // 独占或写文件锁定。
40 #define F_UNLCK          2      // 文件解锁。
41
42 /* Once again - not implemented, but ... */
    /* 同样 - 也还没有实现, 但是... */
    // 文件锁定操作数据结构。描述了受影响文件段的类型(l_type)、开始偏移(l_whence)、
    // 相对偏移(l_start)、锁定长度(l_len)和实施锁定的进程 id。
43 struct flock {
44     short l_type;           // 锁定类型 (F_RDLCK, F_WRLCK, F_UNLCK)。
45     short l_whence;        // 开始偏移 (SEEK_SET, SEEK_CUR 或 SEEK_END)。
46     off_t l_start;         // 阻塞锁定的开始处。相对偏移 (字节数)。
47     off_t l_len;           // 阻塞锁定的大小; 如果是 0 则为到文件末尾。
48     pid_t l_pid;           // 加锁的进程 id。
49 };
50
    // 以下是使用上述标志或命令的函数原型。
    // 创建新文件或重写一个已存在文件。
    // 参数 filename 是欲创建文件的文件名, mode 是创建文件的属性 (见 include/sys/stat.h)。
51 extern int creat(const char * filename, mode_t mode);
    // 文件句柄操作, 会影响文件的打开。
    // 参数 fildes 是文件句柄, cmd 是操作命令, 见上面 23--30 行。该函数可有以下几种形式:
    // int fcntl(int fildes, int cmd);
    // int fcntl(int fildes, int cmd, long arg);
    // int fcntl(int fildes, int cmd, struct flock *lock);
52 extern int fcntl(int fildes, int cmd, ...);
    // 打开文件。在文件与文件句柄之间建立联系。
    // 参数 filename 是欲打开文件的文件名, flags 是上面 7-17 行上的标志的组合。
53 extern int open(const char * filename, int flags, ...);
54
55 #endif
56

```

14.7 signal.h 文件

14.7.1 功能描述

信号提供了一种处理异步事件的方法。信号也被称为是一种软中断。通过向一个进程发送信号，我们可以控制进程的执行状态（暂停、继续或终止）。本文件定义了内核中使用的所有信号的名称和基本操作函数。其中最为重要的函数是改变指定信号处理方式的函数 `signal()` 和 `sigaction()`。

从本文件中可以看出，Linux 内核实现了 POSIX.1 所要求的所有 20 个信号。因此我们可以说 Linux 在一开始设计时就完全考虑到与标准的兼容性了。具体函数的实现见程序 `kernel/signal.c`。

14.7.2 文件注释

程序 14-6 linux/include/signal.h

```

1 #ifndef SIGNAL_H
2 #define SIGNAL_H
3
4 #include <sys/types.h> // 类型头文件。定义了基本的系统数据类型。
5
6 typedef int sig_atomic_t; // 定义信号原子操作类型。
7 typedef unsigned int sigset_t; /* 32 bits */ // 定义信号集类型。
8
9 #define NSIG 32 // 定义信号种类 -- 32 种。
10 #define NSIG NSIG // NSIG = _NSIG
11
12 // 以下这些是 Linux 0.11 内核中定义的信号。其中包括了 POSIX.1 要求的所有 20 个信号。
13 #define SIGHUP 1 // Hang Up -- 挂断控制终端或进程。
14 #define SIGINT 2 // Interrupt -- 来自键盘的中断。
15 #define SIGQUIT 3 // Quit -- 来自键盘的退出。
16 #define SIGILL 4 // Illeagle -- 非法指令。
17 #define SIGTRAP 5 // Trap -- 跟踪断点。
18 #define SIGABRT 6 // Abort -- 异常结束。
19 #define SIGIOT 6 // IO Trap -- 同上。
20 #define SIGUNUSED 7 // Unused -- 没有使用。
21 #define SIGFPE 8 // FPE -- 协处理器出错。
22 #define SIGKILL 9 // Kill -- 强迫进程终止。
23 #define SIGUSR1 10 // User1 -- 用户信号 1，进程可使用。
24 #define SIGSEGV 11 // Segment Violation -- 无效内存引用。
25 #define SIGUSR2 12 // User2 -- 用户信号 2，进程可使用。
26 #define SIGPIPE 13 // Pipe -- 管道写出错，无读者。
27 #define SIGALRM 14 // Alarm -- 实时定时器报警。
28 #define SIGTERM 15 // Terminate -- 进程终止。
29 #define SIGSTKFLT 16 // Stack Fault -- 栈出错（协处理器）。
30 #define SIGCHLD 17 // Child -- 子进程停止或被终止。
31 #define SIGCONT 18 // Continue -- 恢复进程继续执行。
32 #define SIGSTOP 19 // Stop -- 停止进程的执行。
33 #define SIGTSTP 20 // TTY Stop -- tty 发出停止进程，可忽略。
34 #define SIGTTIN 21 // TTY In -- 后台进程请求输入。
35 #define SIGTTOU 22 // TTY Out -- 后台进程请求输出。
36
37 /* Ok, I haven't implemented sigactions, but trying to keep headers POSIX */

```

```

/* OK, 我还没有实现 sigactions 的编制, 但在头文件中仍希望遵守 POSIX 标准 */
// 上面原注释已经过时, 因为在 0.11 内核中已经实现了 sigaction().
37 #define SA_NOCLDSTOP    1                // 当子进程处于停止状态, 就不对 SIGCHLD 处理。
38 #define SA_NOMASK       0x40000000      // 不阻止在指定的信号处理程序中再收到该信号。
39 #define SA_ONESHOT       0x80000000      // 信号句柄一旦被调用过就恢复到默认处理句柄。
40
// 以下常量用于 sigprocmask(how, )-- 改变阻塞信号集(屏蔽码)。用于改变该函数的行为。
41 #define SIG_BLOCK        0 /* for blocking signals */ // 在阻塞信号集中加上给定信号。
42 #define SIG_UNBLOCK      1 /* for unblocking signals */ // 从阻塞信号集中删除指定信号。
43 #define SIG_SETMASK      2 /* for setting the signal mask */ // 设置阻塞信号集。
44
// 以下两个常数符号都表示指向无返回值的函数指针, 且都有一个 int 整型参数。这两个指针
// 值是逻辑上讲实际上不可能出现的函数地址值。可作为下面 signal 函数的第二个参数。用
// 于告知内核, 让内核处理信号或忽略对信号的处理。使用方法参见 kernel/signal.c 程序,
// 第 94—96 行。
45 #define SIG_DFL          ((void (*)(int))0) /* default signal handling */
// 默认信号处理程序(信号句柄)。
46 #define SIG_IGN          ((void (*)(int))1) /* ignore signal */
// 忽略信号的处理程序。
47
// 下面是 sigaction 的数据结构。
// sa_handler 是对应某信号指定要采取的行动。可以用上面的 SIG_DFL, 或 SIG_IGN 来忽略该
// 信号, 也可以是指向处理该信号函数的一个指针。
// sa_mask 给出了对信号的屏蔽码, 在信号程序执行时将阻塞对这些信号的处理。
// sa_flags 指定改变信号处理过程的信号集。它是由 37—39 行的位标志定义的。
// sa_restorer 是恢复函数指针, 由函数库 libc 提供, 用于清理用户态堆栈。参见 signal.c。
// 另外, 引起触发信号处理的信号也将被阻塞, 除非使用了 SA_NOMASK 标志。
48 struct sigaction {
49     void (*sa_handler)(int);
50     sigset_t sa_mask;
51     int sa_flags;
52     void (*sa_restorer)(void);
53 };
54
// 下面 signal 函数用于为信号_sig 安装一新的信号处理程序(信号句柄), 与 sigaction()
// 类似。该函数含有两个参数: 指定需要捕获的信号_sig; 具有一个参数且无返回值的函数指针
// _func。该函数返回值也是具有一个 int 参数(最后一个(int))且无返回值的函数指针, 它是
// 处理该信号的原处理句柄。
55 void (*signal(int_sig, void (*_func)(int)))(int);
// 下面两函数用于发送信号。kill() 用于向任何进程或进程组发送信号。raise() 用于向当前进
// 程自身发送信号。其作用等价于 kill(getpid(), sig)。参见 kernel/exit.c, 60 行。
56 int raise(int sig);
57 int kill(pid_t pid, int sig);
// 在进程的任务结构中, 除有一个以比特位表示当前进程待处理的 32 位信号字段 signal 以外,
// 还有一个同样以比特位表示的用于屏蔽进程当前阻塞信号集(屏蔽信号集)的字段 blocked,
// 也是 32 位, 每个比特代表一个对应的阻塞信号。修改进程的屏蔽信号集可以阻塞或解除阻塞
// 所指定的信号。以下五个函数就是用于操作进程屏蔽信号集, 虽然简单实现起来很简单, 但
// 本版本内核中还未实现。
// 函数 sigaddset() 和 sigdelset() 用于对信号集中的信号进行增、删修改。sigaddset() 用
// 于向 mask 指向的信号集中增加指定的信号 signo。sigdelset 则反之。函数 sigemptyset() 和
// sigfillset() 用于初始化进程屏蔽信号集。每个程序在使用信号集前, 都需要使用这两个函
// 数之一对屏蔽信号集进行初始化。sigemptyset() 用于清空屏蔽的所有信号, 也即响应所有的
// 信号。sigfillset() 向信号集中置入所有信号, 也即屏蔽所有信号。当然 SIGINT 和 SIGSTOP

```

```

// 是不能被屏蔽的。
// sigismember() 用于测试一个指定信号是否在信号集中 (1 - 是, 0 - 不是, -1 - 出错)。
58 int sigaddset(sigset_t *mask, int signo);
59 int sigdelset(sigset_t *mask, int signo);
60 int sigemptyset(sigset_t *mask);
61 int sigfillset(sigset_t *mask);
62 int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */
// 对 set 中的信号进行检测, 看是否有挂起的信号。在 set 中返回进程中当前被阻塞的信号集。
63 int sigpending(sigset_t *set);
// 下面函数用于改变进程目前被阻塞的信号集 (信号屏蔽码)。若 oldset 不是 NULL, 则通过其
// 返回进程当前屏蔽信号集。若 set 指针不是 NULL, 则根据 how (41-43 行) 指示修改进程屏蔽
// 信号集。
64 int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
// 下面函数用 sigmask 临时替换进程的信号屏蔽码, 然后暂停该进程直到收到一个信号。若捕捉
// 到某一信号并从该信号处理程序中返回, 则该函数也返回, 并且信号屏蔽码会恢复到调用调用
// 前的值。
65 int sigsuspend(sigset_t *sigmask);
// sigaction() 函数用于改变进程在收到指定信号时所采取的行动, 即改变信号的处理句柄能。
// 参见对 kernel/signal.c 程序的说明。
66 int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
67
68 #endif /* _SIGNAL_H */
69

```

14.8 stdarg.h 文件

14.8.1 功能描述

C 语言的最大特点之一是允许编程人员自定义参数数目可变的函数。为了访问这些可变参数列表中的参数, 就需要用到 stdarg.h 文件中的宏。stdarg.h 头文件是 C 标准化组织根据 BSD 系统的 varargs.h 文件修改而成。

stdarg.h 是标准参数头文件。它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(va_start, va_arg 和 va_end), 用于 vsprintf、vprintf、vfprintf 函数。在阅读该文件时, 需要首先理解变参函数的使用方法, 可参见 kernel/vsprintf.c 列表后的说明。

14.8.2 代码注释

程序 14-7 linux/include/stdarg.h

```

1 #ifndef _STDARG_H
2 #define _STDARG_H
3
4 typedef char *va_list; // 定义 va_list 是一个字符指针类型。
5
6 /* Amount of space required in an argument list for an arg of type TYPE.
7  TYPE may alternatively be an expression whose type is used. */
8 /* 下面给出了类型为 TYPE 的 arg 参数列表所要求的空间容量。
9  TYPE 也可以是使用该类型的一个表达式 */
10
11 // 下面这句定义了取整后的 TYPE 类型的字节长度值。是 int 长度(4)的倍数。
12 #define __va_rounded_size(TYPE) \

```

```

10  (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
11
    // 下面这个宏初始化指针 AP，使其指向传给函数的可变参数表的第一个参数。
    // 在第一次调用 va_arg 或 va_end 之前，必须首先调用 var_start 宏。参数 LASTARG 是函数定义
    // 中最右边参数的标识符，即'...'左边的一个标识符。AP 是可变参数表参数指针，LASTARG 是
    // 最后一个指定的参数。&(LASTARG)用于取其地址（即其指针），并且该指针是字符类型。加上
    // LASTARG 的宽度值后 AP 就是可变参数表中第一个参数的指针。该宏没有返回值。
    // 第 17 行上的函数 __builtin_saveregs() 是在 gcc 的库程序 libgcc2.c 中定义的，用于保存
    // 寄存器。相关说明参见 gcc 手册“Target Description Macros”章中“Implementing the
    // Varargs Macros”小节。
12 #ifndef __sparc__
13 #define va_start(AP, LASTARG) \
14     (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
15 #else
16 #define va_start(AP, LASTARG) \
17     (__builtin_saveregs (), \
18     AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
19 #endif
20
    // 下面该宏用于被调用函数完成一次正常返回。va_end 可以修改 AP 使其在重新调用
    // va_start 之前不能被使用。va_end 必须在 va_arg 读完所有的参数后再被调用。
21 void va_end (va_list);          /* Defined in gnulib */ /* 在 gnulib 中定义 */
22 #define va_end(AP)
23
    // 下面宏用于扩展表达式使其与下一个被传递参数具有相同的类型和值。
    // va_arg 宏会扩展成函数参数列表中下一个参数的类型和值。AP 应该与 va_start 初始化的
    // va_list AP 相同。每次调用 va_arg 时都会修改 AP，使得下一个参数的值被返回。TYPE 是
    // 一个类型名。在 va_start 初始化之后，第 1 次调用 va_arg 会返回 LASTARG 指定参数后的
    // 参数值。随后的调用会返回随后参数的值。
24 #define va_arg(AP, TYPE) \
25     (AP += __va_rounded_size (TYPE), \
26     *((TYPE *) (AP - __va_rounded_size (TYPE))))
27
28 #endif /* _STDARG_H */
29

```

14.9 stddef.h 文件

14.9.1 功能描述

stddef.h 头文件的名称也是有 C 标准化组织 (X3J11) 创建的，含义是标准 (std) 定义 (def)。主要用于存放一些“标准定义”。另外一个内容容易混淆的头文件是 stdlib.h，也是由标准化组织建立的。stdlib.h 主要用来声明一些不与其他头文件类型相关的各种函数。但这两个头文件中的内容常常让人搞不清哪些声明在哪个头文件中。

标准化组织中的一些成员认为在那些不能完全支持标准 C 库的独立环境中，C 语言也应该成为一种有用的编程语言。对于一个独立环境，C 标准要求其提供 C 语言的所有属性，而对于标准 C 库来说，这样的实现仅需提供支持 4 个头文件中的功能：float.h、limits.h、stdarg.h 和 stddef.h。这个要求明确了 stddef.h 文件应该包含些什么内容，而其他三个头文件基本上用于较为特殊的方面：

float.h 描述浮点表示特性；

limits.h 描述整型表示特性；

stdarg.h 提供用于访问可变参数列表的宏定义。

而独立环境中使用的任何其他类型或宏定义都应该放在 `stddef.h` 文件中。但是后来的组织成员则放宽了这些限制，导致有些定义在多个头文件中出现。例如，宏定义 `NULL` 还出现在其他 4 个头文件中。因此，为了防止冲突，`stddef.h` 文件中在定义 `NULL` 之前首先使用 `undef` 指令取消原先的定义（第 14 行）。

在本文件中定义的类型和宏还有一个共同点：这些定义曾经试图被包含在 C 语言的特性中，但后来由于各种编译器都以各自的方式定义这些信息，很难编写出能取代所有这些定义的代码来，因此就放弃了。

在 Linux 0.11 内核中很少使用该文件。

14.9.2 代码注释

程序 14-8 linux/include/stddef.h

```

1 #ifndef \_STDDEF\_H
2 #define \_STDDEF\_H
3
4 #ifndef \_PTRDIFF\_T
5 #define \_PTRDIFF\_T
6 typedef long ptrdiff\_t;           // 两个指针相减结果的类型。
7 #endif
8
9 #ifndef \_SIZE\_T
10 #define \_SIZE\_T
11 typedef unsigned long size\_t;    // sizeof 返回的类型。
12 #endif
13
14 #undef NULL
15 #define NULL ((void *)0)          // 空指针。
16
17 // 下面定义了一个计算某成员在类型中偏移位置的宏。使用该宏可以确定一个成员（字段）在
18 // 包含它的结构类型中从结构开始处算起的字节偏移量。宏的结果是类型为 size_t 的整数常
19 // 数表达式。这里是一个技巧用法。((TYPE *)0)是将一个整数 0 类型投射（type cast）成数
20 // 据对象指针类型，然后在该结果上进行运算。
21 #define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE *)0)->MEMBER)
22
23 #endif

```

14.10 string.h 文件

14.10.1 功能描述

该头文件中以内嵌函数的形式定义了所有字符串操作函数，为了提高执行速度使用了内嵌汇编程序。另外，在开始处还定义了一个 `NULL` 宏和一个 `SIZE_T` 类型。

在标准 C 库中也提供同样名称的头文件，但函数实现是在标准 C 库中，并且其相应的头文件中只包

含相关函数的声明。而对于下面列出的 string.h 文件，Linus 虽然给出每个函数的实现，但是每个函数都有 'extern' 和 'inline' 关键词前缀，即定义的都是一些内联函数。因此对于包含这个头文件的程序，若由于某种原因所使用的内联函数不能被嵌入调用代码中就会使用内核函数库 lib/目录下定义的同名函数，参见 lib/string.c 程序。在那个 string.c 中，程序首先将 'extern' 和 'inline' 等定义为空，再包含 string.h 头文件，因此，string.c 程序中实际上包含了 string.h 头文件中声明函数的另一个实现代码。

14.10.2 代码注释

程序 14-9 linux/include/string.h

```

1  #ifndef STRING_H
2  #define STRING_H
3
4  #ifndef NULL
5  #define NULL ((void *) 0)
6  #endif
7
8  #ifndef SIZE_T
9  #define SIZE_T
10 typedef unsigned int size_t;
11 #endif
12
13 extern char * strerror(int errno);
14
15 /*
16  * This string-include defines all string functions as inline
17  * functions. Use gcc. It also assumes ds=es=data space, this should be
18  * normal. Most of the string-functions are rather heavily hand-optimized,
19  * see especially strtok, strstr, str[c]spn. They should work, but are not
20  * very easy to understand. Everything is done entirely within the register
21  * set, making the functions fast and clean. String instructions have been
22  * used through-out, making for "slightly" unclear code :-)
23  *
24  * (C) 1991 Linus Torvalds
25  */
26
27 /*
28  * 这个字符串头文件以内嵌函数的形式定义了所有字符串操作函数。使用 gcc 时，同时
29  * 假定了 ds=es=数据空间，这应该是常规的。绝大多数字符串函数都是经手工进行大量
30  * 优化的，尤其是函数 strtok、strstr、str[c]spn。它们应该能正常工作，但却不是那
31  * 么容易理解。所有的操作基本上都是使用寄存器集来完成的，这使得函数即快又整洁。
32  * 所有地方都使用了字符串指令，这又使得代码“稍微”难以理解☺
33  *
34  * (C) 1991 Linus Torvalds
35  */
36
37 // 将一个字符串(src)拷贝到另一个字符串(dest)，直到遇到 NULL 字符后停止。
38 // 参数: dest - 目的字符串指针, src - 源字符串指针。
39 // %0 - esi(src), %1 - edi(dest)。
40 extern inline char * strcpy(char * dest, const char *src)
41 {
42     __asm__ ( "cld\n"                // 清方向位。
43              "l:|t lodsb|n|t"        // 加载 DS:[esi]处 1 字节→al, 并更新 esi。
44              "stosb|n|t"             // 存储字节 al→ES:[edi], 并更新 edi。

```

```

32     "testb %%al, %%al\n\t"        // 刚存储的字节是 0?
33     "jne 1b"                      // 不是则向后跳转到标号 1 处, 否则结束。
34     :: "S" (src), "D" (dest): "si", "di", "ax";
35 return dest;                      // 返回目的字符串指针。
36 }
37
//// 拷贝源字符串 count 个字节到目的字符串。
// 如果源串长度小于 count 个字节, 就附加空字符(NULL)到目的字符串。
// 参数: dest - 目的字符串指针, src - 源字符串指针, count - 拷贝字节数。
// %0 - esi(src), %1 - edi(dest), %2 - ecx(count)。
38 extern inline char * strncpy(char * dest, const char * src, int count)
39 {
40     __asm__ ("cld\n\t"            // 清方向位。
41             "l: \tdecl %2\n\t"    // 寄存器 ecx-- (count--)。
42             "js 2f\n\t"          // 如果 count<0 则向前跳转到标号 2, 结束。
43             "lodsb\n\t"          // 取 ds:[esi]处 1 字节→al, 并且 esi++。
44             "stosb\n\t"          // 存储该字节→es:[edi], 并且 edi++。
45             "testb %%al, %%al\n\t" // 该字节是 0?
46             "jne 1b\n\t"          // 不是, 则向前跳转到标号 1 处继续拷贝。
47             "rep\n\t"            // 否则, 在目的串中存放剩余个数的空字符。
48             "stosb\n\t"
49             "2:"
50             :: "S" (src), "D" (dest), "c" (count): "si", "di", "ax", "cx");
51 return dest;                      // 返回目的字符串指针。
52 }
53
//// 将源字符串拷贝到目的字符串的末尾处。
// 参数: dest - 目的字符串指针, src - 源字符串指针。
// %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1)。
54 extern inline char * strcat(char * dest, const char * src)
55 {
56     __asm__ ("cld\n\t"            // 清方向位。
57             "repne\n\t"          // 比较 al 与 es:[edi]字节, 并更新 edi++,
58             "scasb\n\t"          // 直到找到目的串中是 0 的字节, 此时 edi 已指向后 1 字节。
59             "decl %1\n\t"        // 让 es:[edi]指向 0 值字节。
60             "l: \tlodsb\n\t"     // 取源字符串字节 ds:[esi]→al, 并 esi++。
61             "stosb\n\t"          // 将该字节存到 es:[edi], 并 edi++。
62             "testb %%al, %%al\n\t" // 该字节是 0?
63             "jne 1b"            // 不是, 则向后跳转到标号 1 处继续拷贝, 否则结束。
64             :: "S" (src), "D" (dest), "a" (0), "c" (0xffffffff): "si", "di", "ax", "cx");
65 return dest;                      // 返回目的字符串指针。
66 }
67
//// 将源字符串的 count 个字节复制到目的字符串的末尾处, 最后添一空字符。
// 参数: dest - 目的字符串, src - 源字符串, count - 欲复制的字节数。
// %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1), %4 - (count)。
68 extern inline char * strncat(char * dest, const char * src, int count)
69 {
70     __asm__ ("cld\n\t"            // 清方向位。
71             "repne\n\t"          // 比较 al 与 es:[edi]字节, edi++。
72             "scasb\n\t"          // 直到找到目的串的末端 0 值字节。
73             "decl %1\n\t"        // edi 指向该 0 值字节。
74             "movl %4, %3\n\t"    // 欲复制字节数→ecx。

```

```

75     "1:\tdecl %3\n\t"           // ecx-- (从 0 开始计数)。
76     "js 2f\n\t"                 // ecx < 0 ?, 是则向前跳转到标号 2 处。
77     "lodsb\n\t"                 // 否则取 ds:[esi]处的字节→al, esi++。
78     "stosb\n\t"                 // 存储到 es:[edi]处, edi++。
79     "testb %%al, %%al\n\t"      // 该字节值为 0?
80     "jne 1b\n\t"                 // 不是则向后跳转到标号 1 处, 继续复制。
81     "2:\txorl %2, %2\n\t"       // 将 al 清零。
82     "stosb"                     // 存到 es:[edi]处。
83     :: "S" (src), "D" (dest), "a" (0), "c" (0xffffffff), "g" (count)
84     : "si", "di", "ax", "cx");
85 return dest;                     // 返回目的字符串指针。
86 }
87
//// 将一个字符串与另一个字符串进行比较。
// 参数: cs - 字符串 1, ct - 字符串 2。
// %0 - eax(__res) 返回值, %1 - edi(cs) 字符串 1 指针, %2 - esi(ct) 字符串 2 指针。
// 返回: 如果串 1 > 串 2, 则返回 1; 串 1 = 串 2, 则返回 0; 串 1 < 串 2, 则返回 -1。
// 第 90 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中, 以便于高效访问和操作。
// 这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册“指定寄存器中的变量”。
88 extern inline int strcmp(const char * cs, const char * ct)
89 {
90     register int __res __asm__ ("ax"); // __res 是寄存器变量(eax)。
91     __asm__ ("cld\n\t"                // 清方向位。
92             "1:\tlodsb\n\t"           // 取字符串 2 的字节 ds:[esi]→al, 并且 esi++。
93             "scasb\n\t"               // al 与字符串 1 的字节 es:[edi]作比较, 并且 edi++。
94             "jne 2f\n\t"               // 如果不相等, 则向前跳转到标号 2。
95             "testb %%al, %%al\n\t"    // 该字节是 0 值字节吗(字符串结尾)?
96             "jne 1b\n\t"               // 不是, 则向后跳转到标号 1, 继续比较。
97             "xorl %%eax, %%eax\n\t"    // 是, 则返回值 eax 清零,
98             "jmp 3f\n\t"               // 向前跳转到标号 3, 结束。
99             "2:\tmovl $1, %%eax\n\t"   // eax 中置 1。
100            "jl 3f\n\t"                 // 若前面比较中串 2 字符 < 串 1 字符, 则返回正值结束。
101            "negl %%eax\n\t"            // 否则 eax = -eax, 返回负值, 结束。
102            "3:"
103            : "=a" (__res): "D" (cs), "S" (ct): "si", "di");
104 return __res;                         // 返回比较结果。
105 }
106
//// 字符串 1 与字符串 2 的前 count 个字符进行比较。
// 参数: cs - 字符串 1, ct - 字符串 2, count - 比较的字符数。
// %0 - eax(__res) 返回值, %1 - edi(cs) 串 1 指针, %2 - esi(ct) 串 2 指针, %3 - ecx(count)。
// 返回: 如果串 1 > 串 2, 则返回 1; 串 1 = 串 2, 则返回 0; 串 1 < 串 2, 则返回 -1。
107 extern inline int strncmp(const char * cs, const char * ct, int count)
108 {
109     register int __res __asm__ ("ax"); // __res 是寄存器变量(eax)。
110     __asm__ ("cld\n\t"                // 清方向位。
111             "1:\tdecl %3\n\t"          // count--。
112             "js 2f\n\t"                // 如果 count < 0, 则向前跳转到标号 2。
113             "lodsb\n\t"                // 取串 2 的字符 ds:[esi]→al, 并且 esi++。
114             "scasb\n\t"                // 比较 al 与串 1 的字符 es:[edi], 并且 edi++。
115             "jne 3f\n\t"                // 如果不相等, 则向前跳转到标号 3。
116             "testb %%al, %%al\n\t"     // 该字符是 NULL 字符吗?
117             "jne 1b\n\t"                // 不是, 则向后跳转到标号 1, 继续比较。

```

```

118     "2:\txorl %%eax, %%eax\n\t" // 是 NULL 字符, 则 eax 清零 (返回值)。
119     "jmp 4f\n\t" // 向前跳转到标号 4, 结束。
120     "3:\tmovl $1, %%eax\n\t" // eax 中置 1。
121     "jl 4f\n\t" // 如果前面比较中串 2 字符 < 串 1 字符, 则返回 1 结束。
122     "negl %%eax\n\t" // 否则 eax = -eax, 返回负值, 结束。
123     "4:"
124     : "=a" (__res): "D" (cs), "S" (ct), "c" (count): "si", "di", "cx");
125 return __res; // 返回比较结果。
126 }
127
128 // 在字符串中寻找第一个匹配的字符。
129 // 参数: s - 字符串, c - 欲寻找的字符。
130 // %0 - eax(__res), %1 - esi(字符串指针 s), %2 - eax(字符 c)。
131 // 返回: 返回字符串中第一次出现匹配字符的指针。若没有找到匹配的字符, 则返回空指针。
128 extern inline char * strchr(const char * s, char c)
129 {
130     register char * __res __asm__ ("ax"); // __res 是寄存器变量(eax)。
131     __asm__ ("cld\n\t" // 清方向位。
132             "movb %%al, %%ah\n\t" // 将欲比较字符移到 ah。
133             "l:\tlodsb\n\t" // 取字符串中字符 ds:[esi]→al, 并且 esi++。
134             "cmpb %%ah, %%al\n\t" // 字符串中字符 al 与指定字符 ah 相比较。
135             "je 2f\n\t" // 若相等, 则向前跳转到标号 2 处。
136             "testb %%al, %%al\n\t" // al 中字符是 NULL 字符吗? (字符串结尾?)
137             "jne 1b\n\t" // 若不是, 则向后跳转到标号 1, 继续比较。
138             "movl $1, %1\n\t" // 是, 则说明没有找到匹配字符, esi 置 1。
139             "2:\tmovl %1, %0\n\t" // 将指向匹配字符后一个字节处的指针值放入 eax
140             "decl %0" // 将指针调整为指向匹配的字符。
141             : "=a" (__res): "S" (s), "0" (c): "si");
142 return __res; // 返回指针。
143 }
144
145 // 寻找字符串中指定字符最后一次出现的地方。(反向搜索字符串)
146 // 参数: s - 字符串, c - 欲寻找的字符。
147 // %0 - edx(__res), %1 - edx(0), %2 - esi(字符串指针 s), %3 - eax(字符 c)。
148 // 返回: 返回字符串中最后一次出现匹配字符的指针。若没有找到匹配的字符, 则返回空指针。
145 extern inline char * strrchr(const char * s, char c)
146 {
147     register char * __res __asm__ ("dx"); // __res 是寄存器变量(edx)。
148     __asm__ ("cld\n\t" // 清方向位。
149             "movb %%al, %%ah\n\t" // 将欲寻找的字符移到 ah。
150             "l:\tlodsb\n\t" // 取字符串中字符 ds:[esi]→al, 并且 esi++。
151             "cmpb %%ah, %%al\n\t" // 字符串中字符 al 与指定字符 ah 作比较。
152             "jne 2f\n\t" // 若不相等, 则向前跳转到标号 2 处。
153             "movl %%esi, %0\n\t" // 将字符指针保存到 edx 中。
154             "decl %0\n\t" // 指针后退一位, 指向字符串中匹配字符处。
155             "2:\ttstb %%al, %%al\n\t" // 比较的字符是 0 吗 (到字符串尾)?
156             "jne 1b" // 不是则向后跳转到标号 1 处, 继续比较。
157             : "=d" (__res): "0" (0), "S" (s), "a" (c): "ax", "si");
158 return __res; // 返回指针。
159 }
160
161 // 在字符串 1 中寻找第 1 个字符序列, 该字符序列中的任何字符都包含在字符串 2 中。
162 // 参数: cs - 字符串 1 指针, ct - 字符串 2 指针。

```

```

// %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi(串 1 指针 cs), %4 - (串 2 指针 ct)。
// 返回字符串 1 中包含字符串 2 中任何字符的首个字符序列的长度值。
161 extern inline int strspn(const char * cs, const char * ct)
162 {
163     register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
164     __asm__("cld\n\t" // 清方向位。
165             "movl %4, %%edi\n\t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
166             "repne\n\t" // 比较 al(0)与串 2 中的字符(es:[edi])，并 edi++。
167             "scasb\n\t" // 如果不相等就继续比较(ecx 逐步递减)。
168             "notl %%ecx\n\t" // ecx 中每位取反。
169             "decl %%ecx\n\t" // ecx--，得串 2 的长度值。
170             "movl %%ecx, %%edx\n\t" // 将串 2 的长度值暂放入 edx 中。
171             "l:\t\tlods\n\t" // 取串 1 字符 ds:[esi]→al，并且 esi++。
172             "testb %%al, %%al\n\t" // 该字符等于 0 值吗(串 1 结尾)?
173             "je 2f\n\t" // 如果是，则向前跳转到标号 2 处。
174             "movl %4, %%edi\n\t" // 取串 2 头指针放入 edi 中。
175             "movl %%edx, %%ecx\n\t" // 再将串 2 的长度值放入 ecx 中。
176             "repne\n\t" // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
177             "scasb\n\t" // 如果不相等就继续比较。
178             "je 1b\n\t" // 如果相等，则向后跳转到标号 1 处。
179             "2:\t\tdecl %0" // esi--，指向最后一个包含在串 2 中的字符。
180             : "=S" (__res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
181             : "ax", "cx", "dx", "di");
182     return __res-cs; // 返回字符序列的长度值。
183 }
184
//// 寻找字符串 1 中不包含字符串 2 中任何字符的首个字符序列。
// 参数: cs - 字符串 1 指针, ct - 字符串 2 指针。
// %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi(串 1 指针 cs), %4 - (串 2 指针 ct)。
// 返回字符串 1 中不包含字符串 2 中任何字符的首个字符序列的长度值。
185 extern inline int strcspn(const char * cs, const char * ct)
186 {
187     register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
188     __asm__("cld\n\t" // 清方向位。
189             "movl %4, %%edi\n\t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
190             "repne\n\t" // 比较 al(0)与串 2 中的字符(es:[edi])，并 edi++。
191             "scasb\n\t" // 如果不相等就继续比较(ecx 逐步递减)。
192             "notl %%ecx\n\t" // ecx 中每位取反。
193             "decl %%ecx\n\t" // ecx--，得串 2 的长度值。
194             "movl %%ecx, %%edx\n\t" // 将串 2 的长度值暂放入 edx 中。
195             "l:\t\tlods\n\t" // 取串 1 字符 ds:[esi]→al，并且 esi++。
196             "testb %%al, %%al\n\t" // 该字符等于 0 值吗(串 1 结尾)?
197             "je 2f\n\t" // 如果是，则向前跳转到标号 2 处。
198             "movl %4, %%edi\n\t" // 取串 2 头指针放入 edi 中。
199             "movl %%edx, %%ecx\n\t" // 再将串 2 的长度值放入 ecx 中。
200             "repne\n\t" // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
201             "scasb\n\t" // 如果不相等就继续比较。
202             "jne 1b\n\t" // 如果不相等，则向后跳转到标号 1 处。
203             "2:\t\tdecl %0" // esi--，指向最后一个包含在串 2 中的字符。
204             : "=S" (__res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
205             : "ax", "cx", "dx", "di");
206     return __res-cs; // 返回字符序列的长度值。
207 }

```

```

208 // 在字符串 1 中寻找首个包含在字符串 2 中的任何字符。
209 // 参数: cs - 字符串 1 的指针, ct - 字符串 2 的指针。
210 // %0 -esi(__res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(串 1 指针 cs), %4 -(串 2 指针 ct)。
211 // 返回字符串 1 中首个包含字符串 2 中字符的指针。
212 extern inline char * strpbrk(const char * cs, const char * ct)
213 {
214     register char * __res __asm__( "si" ); // __res 是寄存器变量(esi)。
215     __asm__( "cld\n\t" // 清方向位。
216             "movl %4, %%edi\n\t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
217             "repne\n\t" // 比较 al(0) 与串 2 中的字符 (es:[edi]), 并 edi++。
218             "scasb\n\t" // 如果不相等就继续比较 (ecx 逐步递减)。
219             "notl %%ecx\n\t" // ecx 中每位取反。
220             "decl %%ecx\n\t" // ecx--, 得串 2 的长度值。
221             "movl %%ecx, %%edx\n\t" // 将串 2 的长度值暂放入 edx 中。
222             "l:\t lodsb\n\t" // 取串 1 字符 ds:[esi]→al, 并且 esi++。
223             "testb %%al, %%al\n\t" // 该字符等于 0 值吗 (串 1 结尾)?
224             "je 2f\n\t" // 如果是, 则向前跳转到标号 2 处。
225             "movl %4, %%edi\n\t" // 取串 2 头指针放入 edi 中。
226             "movl %%edx, %%ecx\n\t" // 再将串 2 的长度值放入 ecx 中。
227             "repne\n\t" // 比较 al 与串 2 中字符 es:[edi], 并且 edi++。
228             "scasb\n\t" // 如果不相等就继续比较。
229             "jne 1b\n\t" // 如果不相等, 则向后跳转到标号 1 处。
230             "decl %0\n\t" // esi--, 指向一个包含在串 2 中的字符。
231             "jmp 3f\n\t" // 向前跳转到标号 3 处。
232             "2:\txorl %0, %0\n\t" // 没有找到符合条件的, 将返回值为 NULL。
233             "3:"
234             : "=S" (__res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
235             : "ax", "cx", "dx", "di" );
236     return __res; // 返回指针值。
237 }
238
239 // 在字符串 1 中寻找首个匹配整个字符串 2 的字符串。
240 // 参数: cs - 字符串 1 的指针, ct - 字符串 2 的指针。
241 // %0 -eax(__res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(串 1 指针 cs), %4 -(串 2 指针 ct)。
242 // 返回: 返回字符串 1 中首个匹配字符串 2 的字符串指针。
243 extern inline char * strstr(const char * cs, const char * ct)
244 {
245     register char * __res __asm__( "ax" ); // __res 是寄存器变量(eax)。
246     __asm__( "cld\n\t" \
247             "movl %4, %%edi\n\t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
248             "repne\n\t" // 比较 al(0) 与串 2 中的字符 (es:[edi]), 并 edi++。
249             "scasb\n\t" // 如果不相等就继续比较 (ecx 逐步递减)。
250             "notl %%ecx\n\t" // ecx 中每位取反。
251             "decl %%ecx\n\t" /* NOTE! This also sets Z if searchstring="" */
252                             /* 注意! 如果搜索串为空, 将设置 Z 标志 */ // 得串 2 的长度值。
253             "movl %%ecx, %%edx\n\t" // 将串 2 的长度值暂放入 edx 中。
254             "l:\t movl %4, %%edi\n\t" // 取串 2 头指针放入 edi 中。
255             "movl %%esi, %%eax\n\t" // 将串 1 的指针复制到 eax 中。
256             "movl %%edx, %%ecx\n\t" // 再将串 2 的长度值放入 ecx 中。
257             "repe\n\t" // 比较串 1 和串 2 字符(ds:[esi], es:[edi]), esi++, edi++。
258             "cmpsb\n\t" // 若对应字符相等就一直比较下去。
259             "je 2f\n\t" /* also works for empty string, see above */

```



```

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

```



```

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

```

```

339     "rep|n|t"           // 重复执行复制 ecx 个字节,
340     "movsb"             // 从 ds:[esi]到 es:[edi], esi++, edi++.
341     :: "c" (n), "S" (src), "D" (dest)
342     : "cx", "si", "di");
343 return dest;           // 返回目的地址。
344 }
345
346 // 内存块移动。同内存块复制, 但考虑移动的方向。
347 // 参数: dest - 复制的目的地址, src - 复制的源地址, n - 复制字节数。
348 // 若 dest<src 则: %0 - ecx(n), %1 - esi(src), %2 - edi(dest)。
349 // 否则: %0 - ecx(n), %1 - esi(src+n-1), %2 - edi(dest+n-1)。
350 // 这样操作是为了防止在复制时错误地重叠覆盖。
351 extern inline void * memmove(void * dest, const void * src, int n)
352 {
353     if (dest<src)
354     {
355         __asm__ ("cld|n|t"           // 清方向位。
356                 "rep|n|t"           // 从 ds:[esi]到 es:[edi], 并且 esi++, edi++,
357                 "movsb"             // 重复执行复制 ecx 字节。
358                 :: "c" (n), "S" (src), "D" (dest)
359                 : "cx", "si", "di");
360     }
361     else
362     {
363         __asm__ ("std|n|t"           // 置方向位, 从末端开始复制。
364                 "rep|n|t"           // 从 ds:[esi]到 es:[edi], 并且 esi--, edi--,
365                 "movsb"             // 复制 ecx 个字节。
366                 :: "c" (n), "S" (src+n-1), "D" (dest+n-1)
367                 : "cx", "si", "di");
368     }
369     return dest;
370 }
371
372 // 比较 n 个字节的两块内存 (两个字符串), 即使遇上 NULL 字节也不停止比较。
373 // 参数: cs - 内存块 1 地址, ct - 内存块 2 地址, count - 比较的字节数。
374 // %0 - eax(__res), %1 - eax(0), %2 - edi(内存块 1), %3 - esi(内存块 2), %4 - ecx(count)。
375 // 返回: 若块 1>块 2 返回 1; 块 1<块 2, 返回-1; 块 1==块 2, 则返回 0。
376 extern inline int memcmp(const void * cs, const void * ct, int count)
377 {
378     register int __res __asm__ ("ax"); // __res 是寄存器变量。
379     __asm__ ("cld|n|t"           // 清方向位。
380             "repe|n|t"          // 如果相等则重复,
381             "cmpsb|n|t"         // 比较 ds:[esi]与 es:[edi]的内容, 并且 esi++, edi++.
382             "je 1f|n|t"         // 如果都相同, 则跳转到标号 1, 返回 0(eax)值
383             "movl $1, %%eax|n|t" // 否则 eax 置 1,
384             "jl 1f|n|t"         // 若内存块 2 内容的值<内存块 1, 则跳转标号 1。
385             "negl %%eax|n|t"     // 否则 eax = -eax。
386             "1:"
387             : "=a" (__res): "0" (0), "D" (cs), "S" (ct), "c" (count)
388             : "si", "di", "cx");
389     return __res;           // 返回比较结果。
390 }
391
392 // 在 n 字节大小的内存块 (字符串) 中寻找指定字符。
393 // 参数: cs - 指定内存块地址, c - 指定的字符, count - 内存块长度。
394 // %0 - edi(__res), %1 - eax(字符 c), %2 - edi(内存块地址 cs), %3 - ecx(字节数 count)。
395 // 返回第一个匹配字符的指针, 如果没有找到, 则返回 NULL 字符。

```

```

379 extern inline void * memchr(const void * cs, char c, int count)
380 {
381     register void * __res __asm__ ("di"); // __res 是寄存器变量。
382     if (!count) // 如果内存块长度==0, 则返回 NULL, 没有找到。
383         return NULL;
384     __asm__ ("cld|n|t" // 清方向位。
385             "repne|n|t" // 如果不相等则重复执行下面语句,
386             "scasb|n|t" // al 中字符与 es:[edi]字符作比较, 并且 edi++,
387             "je 1f|n|t" // 如果相等则向前跳转到标号 1 处。
388             "movl $1, %0|n" // 否则 edi 中置 1。
389             "1:|tdecl %0" // 让 edi 指向找到的字符 (或是 NULL)。
390             : "=D" (__res) : "a" (c), "D" (cs), "c" (count)
391             : "cx");
392     return __res; // 返回字符指针。
393 }
394
395 // 用字符 c 填充指定长度内存块。
396 // 用字符 c 填充 s 指向的内存区域, 共填充 count 字节。
397 // %0 - eax(字符 c), %1 - edi(内存地址), %2 - ecx(字节数 count)。
398 extern inline void * memset(void * s, char c, int count)
399 {
400     __asm__ ("cld|n|t" // 清方向位。
401             "rep|n|t" // 重复 ecx 指定的次数, 执行
402             "stosb" // 将 al 中字符存入 es:[edi]中, 并且 edi++。
403             : : "a" (c), "D" (s), "c" (count)
404             : "cx", "di");
405     return s;
406 }
407 #endif
408

```

14.11 `termios.h` 文件

14.11.1 功能描述

该文件含有终端 I/O 接口定义。包括 `termios` 数据结构和一些对通用终端接口设置的函数原型。这些函数用来读取或设置终端的属性、线路控制、读取或设置波特率以及读取或设置终端前端进程的组 id。虽然这是 Linux 早期的头文件, 但已完全符合目前的 POSIX 标准, 并作了适当的扩展。

在该文件中定义的两个终端数据结构 `termio` 和 `termios` 是分别属于两类 UNIX 系列 (或克隆), `termio` 是在 AT&T 系统 V 中定义的, 而 `termios` 是 POSIX 标准指定的。两个结构基本一样, 只是 `termio` 使用短整数类型定义模式标志集, 而 `termios` 使用长整数定义模式标志集。由于目前这两种结构都在使用, 因此为了兼容性, 大多数系统都同时支持它们。另外, 以前使用的是一类似的 `sgtty` 结构, 目前已基本不用。

14.11.2 代码注释

程序 14-10 `linux/include/termios.h`

```

1 #ifndef TERMIOS\_H
2 #define TERMIOS\_H
3
4 #define TTY\_BUF\_SIZE 1024           // tty 中的缓冲区长度。
5
6 /* 0x54 is just a magic number to make these relatively unique ('T') */
/* 0x54 只是一个魔数，目的是为了使这些常数唯一('T') */
7
8 // tty 设备的 ioctl 调用命令集。ioctl 将命令编码在低位字中。
9 // 下面名称 TC[*] 的含义是 tty 控制命令。
10 // 取相应终端 termios 结构中的信息(参见 tcgetattr())。
11 #define TCGETS 0x5401
12 // 设置相应终端 termios 结构中的信息(参见 tcsetattr(), TCSANOW)。
13 #define TCSETS 0x5402
14 // 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
15 // 会影响输出的情况，就需要使用这种形式(参见 tcsetattr(), TCSADRAIN 选项)。
16 #define TCSETSW 0x5403
17 // 在设置 termios 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
18 // 再设置(参见 tcsetattr(), TCSAFLUSH 选项)。
19 #define TCSETSF 0x5404
20 // 取相应终端 termio 结构中的信息(参见 tcgetattr())。
21 #define TCGETA 0x5405
22 // 设置相应终端 termio 结构中的信息(参见 tcsetattr(), TCSANOW 选项)。
23 #define TCSETA 0x5406
24 // 在设置终端 termio 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
25 // 会影响输出的情况，就需要使用这种形式(参见 tcsetattr(), TCSADRAIN 选项)。
26 #define TCSETAW 0x5407
27 // 在设置 termio 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
28 // 再设置(参见 tcsetattr(), TCSAFLUSH 选项)。
29 #define TCSETAF 0x5408
30 // 等待输出队列处理完毕(空)，如果参数值是 0，则发送一个 break(参见 tcsendbreak(), tcdrain())。
31 #define TCSBRK 0x5409
32 // 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，则挂
33 起
34 // 输入；如果是 3，则重新开启挂起的输入(参见 tcflow())。
35 #define TCXONC 0x540A
36 // 刷新已写输出但还没发送或已收但还没有读数据。如果参数是 0，则刷新(清空)输入队列；如果是 1，
37 // 则刷新输出队列；如果是 2，则刷新输入和输出队列(参见 tcflush())。
38 #define TCFLSH 0x540B
39 // 下面名称 TIOC[*] 的含义是 tty 输入输出控制命令。
40 // 设置终端串行线路专用模式。
41 #define TIOCEXCL 0x540C
42 // 复位终端串行线路专用模式。
43 #define TIOCNXCL 0x540D
44 // 设置 tty 为控制终端。(TIOCNOTTY - 禁止 tty 为控制终端)。
45 #define TIOCSCTTY 0x540E
46 // 读取指定终端设备进程的组 id(参见 tcgetpgrp())。
47 #define TIOCGPRG 0x540F
48 // 设置指定终端设备进程的组 id(参见 tcsetpgrp())。

```

```

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

```

```

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

```



```

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

151 #define CSIZE 0000060 // 字符位宽度屏蔽码。
152 #define CS5 0000000 // 每字符 5 比特位。
153 #define CS6 0000020 // 每字符 6 比特位。
154 #define CS7 0000040 // 每字符 7 比特位。
155 #define CS8 0000060 // 每字符 8 比特位。

```



```

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

```

```

205 #define TCIFLUSH      0          // 清接收到的数据但不读。
206 #define TCOFLUSH      1          // 清已写的输出数据但不传送。
207 #define TCIOFLUSH     2          // 清接收到的数据但不读。清已写的输出数据但不传送。
208
209 /* tcsetattr uses these */      /* tcsetattr() 使用这些符号常数 */
210 #define TCSANOW        0          // 改变立即发生。
211 #define TCSADRAIN      1          // 改变在所有已写的输出被传输之后发生。
212 #define TCSAFLUSH      2          // 改变在所有已写的输出被传输之后并且在所有接收到但
                                   // 还没有读取的数据被丢弃之后发生。
213
214 typedef int speed_t;           // 波特率数值类型。
215
// 以下这些函数在编译环境的函数库 libc.a 中实现，内核中没有。在函数库实现中，这些函数通过
// 调用系统调用 ioctl() 来实现。有关 ioctl() 系统调用，请参见 fs/ioctl.c 程序。
// 返回 termios_p 所指 termios 结构中的接收波特率。
216 extern speed_t cfgetispeed(struct termios *termios_p);
// 返回 termios_p 所指 termios 结构中的发送波特率。
217 extern speed_t cfgetospeed(struct termios *termios_p);
// 将 termios_p 所指 termios 结构中的接收波特率设置为 speed。
218 extern int cfsetispeed(struct termios *termios_p, speed_t speed);
// 将 termios_p 所指 termios 结构中的发送波特率设置为 speed。
219 extern int cfsetospeed(struct termios *termios_p, speed_t speed);
// 等待 fildes 所指对象已写输出数据被传送出去。
220 extern int tcdrain(int fildes);
// 挂起/重启 fildes 所指对象数据的接收和发送。
221 extern int tcflow(int fildes, int action);
// 丢弃 fildes 指定对象所有已写但还没传送以及所有已收到但还没有读取的数据。
222 extern int tcflush(int fildes, int queue_selector);
// 获取与句柄 fildes 对应对象的参数，并将其保存在 termios_p 所指的地方。
223 extern int tcgetattr(int fildes, struct termios *termios_p);
// 如果终端使用异步串行数据传输，则在一定时间内连续传输一系列 0 值比特位。
224 extern int tcsendbreak(int fildes, int duration);
// 使用 termios 结构指针 termios_p 所指的数据，设置与终端相关的参数。
225 extern int tcsetattr(int fildes, int optional_actions,
226                     struct termios *termios_p);
227
228 #endif
229

```

14.11.3 其他信息

14.11.3.1 控制字符 TIME、MIN

在非规范模式输入处理中，输入字符没有被处理成行，因此擦除和终止处理也就不会发生。MIN 和 TIMEDE 的值即用于确定如何处理接收到的字符。

MIN 表示当满足读操作时（也即，当字符返给用户时）需要读取的最少字符数。TIME 是以 1/10 秒计数的定时值，用于猝发和短时期数据传输的超时值。这两个字符的四种组合情况及其相互作用描述如下：

◆ MIN > 0, TIME > 0 的情况：

在这种情况下，TIME 起字符与字符间的定时器作用，并在接收到第 1 个字符后开始起作用。由于它是字符与字符间的定时器，所以在每收到一个字符就会被复位重启。MIN 与 TIME 之间的相互作用如下：一旦收到一个字符，字符间定时器就开始工作。如果在定时器超时（注意定时器每收到一个字符就

会重新开始计时)之前收到了 MIN 个字符,则读操作即被满足。如果在 MIN 个字符被收到之前定时器超时了,就将到此时已收到的字符返回给用户。注意,如果 TIME 超时,则起码有一个接收到的字符将被返回,因为定时器只有在接收到了一个字符之后才开始起作用(计时)。在这种情况下(MIN > 0, TIME > 0),读操作将会睡眠,直到接收到第 1 个字符激活 MIN 与 TIME 机制。如果读到字符数少于已有的字符数,那么定时器将不会被重新激活,因而随后的读操作将被立刻满足。

◆ MIN > 0, TIME = 0 的情况:

在这种情况下,由于 TIME 的值是 0,因此定时器不起作用,只有 MIN 是有意义的。等待的读操作只有当接收到 MIN 个字符时才会被满足(等待着的操作将睡眠直到收到 MIN 个字符)。使用这种情况去读基于记录的终端 IO 的程序将会在读操作中被不确定地(随意地)阻塞。

◆ MIN = 0, TIME > 0 的情况:

在这种情况下,由于 MIN=0,则 TIME 不再起字符间的定时器作用,而是一个读操作定时器,并在读操作一开始就起作用。只要接收到一个字符或者定时器超时就已满足读操作。注意,在这种情况下,如果定时器超时了,将读不到一个字符。如果定时器没有超时,那么只有在读到一个字符之后读操作才会满足。因此在这种情况下,读操作不会无限制地(不确定地)被阻塞,以等待字符。在读操作开始后,如果在 TIME*0.10 秒的时间内没有收到字符,读操作将以收到 0 个字符而返回。

◆ MIN = 0, TIME = 0 的情况:

在这种情况下,读操作会立刻返回。所请求读的字符数或缓冲队列中现有字符数中的最小值将被返回,而不会等待更多的字符被输入缓冲中。

总得来说,在非规范模式下,这两个值是超时定时值和字符计数值。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,然后在以读取 MIN 个字符或者时间 TIME 在读取最后一个字符后超时。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。

14.12 time.h 文件

14.12.1 功能描述

time.h 头文件用于涉及处理时间和日期的函数。在 MINIX 中有一段对时间很有趣的描述:时间的处理较为复杂,比如什么是 GMT(格林威治标准时间,现在是 UTC 时间)、本地时间或其他时间等。尽管主教 Ussher(1581-1656 年)曾经计算过,根据圣经,世界开始之日是公元前 4004 年 10 月 12 日上午 9 点,但在 UNIX 世界里,时间是从 GMT 1970 年 1 月 1 日午夜开始的,在这之前,所有均是空无的和(无效的)。

该文件是标准 C 库中的头文件之一。由于当时 UNIX 操作系统开发者中有一些是业余天文爱好者,所以他们对 UNIX 系统中时间的表示要求特别严格,以至于在 UNIX 类系统中或与标准 C 兼容的系统中有关时间和日期的表示和计算特别复杂。该文件定义了 1 个常数符号(宏)、4 个类型以及一些时间和日期操作转换函数。在 Linux 0.11 内核中,该文件主要为 init/main.c 和 kernel/mktime.c 文件提供 tm 结构类型,用于内核从系统 CMOS 芯片中获取实时时钟信息(日历时间),从而可以设定系统开机时间。开机时间是指从 1970 年 1 月 1 日午夜 0 时起当开机时经过的时间(秒),它将保存在全局变量 startup_time 中供内核所有代码读取。

另外,该文件中给出的一些函数声明均是标准 C 库提供的函数。内核中不包括这些函数。

14.12.2 代码注释

程序 14-11 linux/include/time.h

```

1 #ifndef TIME_H
2 #define TIME_H
3
4 #ifndef TIME_T
5 #define TIME_T
6 typedef long time_t;           // 从 GMT 1970 年 1 月 1 日午夜 0 时起开始计的时间（秒）。
7 #endif
8
9 #ifndef SIZE_T
10 #define SIZE_T
11 typedef unsigned int size_t;
12 #endif
13
14 #define CLOCKS_PER_SEC 100      // 系统时钟滴答频率，100HZ。
15
16 typedef long clock_t;          // 从进程开始系统经过的时钟滴答数。
17
18 struct tm {
19     int tm_sec;                  // 秒数 [0, 59]。
20     int tm_min;                  // 分钟数 [ 0, 59]。
21     int tm_hour;                 // 小时数 [0, 59]。
22     int tm_mday;                 // 1 个月的天数 [0, 31]。
23     int tm_mon;                  // 1 年中月份 [0, 11]。
24     int tm_year;                 // 从 1900 年开始的年数。
25     int tm_wday;                 // 1 星期中的某天 [0, 6]（星期天 =0）。
26     int tm_yday;                 // 1 年中的某天 [0, 365]。
27     int tm_isdst;                // 夏令时标志。正数 - 使用；0 - 没有使用；负数 - 无效。
28 };
29
30 // 以下是有关时间操作的函数原型。
31 // 确定处理器使用时间。返回程序所用处理器时间（滴答数）的近似值。
32 clock_t clock(void);
33 // 取时间（秒数）。返回从 1970.1.1:0:0:0 开始的秒数（称为日历时间）。
34 time_t time(time_t * tp);
35 // 计算时间差。返回时间 time2 与 time1 之间经过的秒数。
36 double difftime(time_t time2, time_t time1);
37 // 将 tm 结构表示的时间转换成日历时间。
38 time_t mktime(struct tm * tp);
39
40 // 将 tm 结构表示的时间转换成字符串。返回指向该串的指针。
41 char * asctime(const struct tm * tp);
42 // 将日历时间转换成字符串形式，如 “Wed Jun 30 21:49:08:1993\n”。
43 char * ctime(const time_t * tp);
44 // 将日历时间转换成 tm 结构表示的 UTC 时间（UTC - 世界时间代码 Universal Time Code）。
45 struct tm * gmtime(const time_t * tp);
46 // 将日历时间转换成 tm 结构表示的指定时区 (Time Zone) 的时间。
47 struct tm * localtime(const time_t * tp);
48 // 将 tm 结构表示的时间利用格式字符串 fmt 转换成最大长度为 smax 的字符串并将结果存储在 s 中。
49 size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);

```

```

// 初始化时间转换信息，使用环境变量 TZ，对 zname 变量进行初始化。
// 在与时区相关的时间转换函数中将自动调用该函数。
40 void tzset(void);
41
42 #endif
43

```

14.13 unistd.h 文件

14.13.1 功能描述

标准符号常数和类型头文件。该文件中定义了很多各种各样的常数和类型，以及一些函数声明。如果在程序中定义了符号 `__LIBRARY__`，则还包括内核系统调用号和内嵌汇编 `_syscall0()` 等。

14.13.2 代码注释

程序 14-12 linux/include/unistd.h

```

1 #ifndef UNISTD_H
2 #define UNISTD_H
3
4 /* ok, this may be a joke, but I'm working on it */
   /* ok, 这也许是个玩笑，但我正在着手处理 */
   // 下面符号常数指出符合 IEEE 标准 1003.1 实现的版本号，是一个整数值。
5 #define POSIX_VERSION 198808L
6
   // chown() 和 fchown() 的使用受限于进程的权限。/* 只有超级用户可以执行 chown (我想...) */
7 #define POSIX_CHOWN_RESTRICTED /* only root can do a chown (I think..) */
   // 长于 (NAME_MAX) 的路径名将产生错误，而不会自动截断。/* 路径名不截断（但是请看内核代码）*/
8 #define POSIX_NO_TRUNC /* no pathname truncation (but see in kernel) */
   // 下面这个符号将定义成字符值，该值将禁止终端对其的处理。/* 禁止象 ^C 这样的字符 */
9 #define POSIX_VDISABLE '\0' /* character to disable things like ^C */
   // 每个进程都有一保存的 set-user-ID 和一保存的 set-group-ID。/* 我们将着手对此进行处理 */
10 /*#define _POSIX_SAVED_IDS */ /* we'll get to this yet */
   // 系统实现支持作业控制。/* 我们还没有支持这项标准，希望很快就行 */
11 /*#define _POSIX_JOB_CONTROL */ /* we aren't there quite yet. Soon hopefully */
12
13 #define STDIN_FILENO 0 // 标准输入文件句柄（描述符）号。
14 #define STDOUT_FILENO 1 // 标准输出文件句柄号。
15 #define STDERR_FILENO 2 // 标准出错文件句柄号。
16
17 #ifndef NULL
18 #define NULL ((void *)0) // 定义空指针。
19 #endif
20
21 /* access */ /* 文件访问 */
   // 以下定义的符号常数用于 access() 函数。
22 #define F_OK 0 // 检测文件是否存在。
23 #define X_OK 1 // 检测是否可执行（搜索）。
24 #define W_OK 2 // 检测是否可写。

```

```

25 #define R_OK      4           // 检测是否可读。
26
27 /* lseek */ /* 文件指针重定位 */
    // 以下符号常数用于 lseek() 和 fcntl() 函数。
28 #define SEEK_SET      0       // 将文件读写指针设置为偏移值。
29 #define SEEK_CUR      1       // 将文件读写指针设置为当前值加上偏移值。
30 #define SEEK_END      2       // 将文件读写指针设置为文件长度加上偏移值。
31
32 /* _SC stands for System Configuration. We don't use them much */
    /* _SC 表示系统配置。我们很少使用 */
    // 下面的符号常数用于 sysconf() 函数。
33 #define SC_ARG_MAX      1     // 最大变量数。
34 #define SC_CHILD_MAX    2     // 子进程最大数。
35 #define SC_CLOCKS_PER_SEC 3   // 每秒滴答数。
36 #define SC_NGROUPS_MAX  4     // 最大组数。
37 #define SC_OPEN_MAX     5     // 最大打开文件数。
38 #define SC_JOB_CONTROL   6     // 作业控制。
39 #define SC_SAVED_IDS     7     // 保存的标识符。
40 #define SC_VERSION       8     // 版本。
41
42 /* more (possibly) configurable things - now pathnames */
    /* 更多的（可能的）可配置参数 - 现在用于路径名 */
    // 下面的符号常数用于 pathconf() 函数。
43 #define PC_LINK_MAX      1     // 连接最大数。
44 #define PC_MAX_CANON      2     // 最大常规文件数。
45 #define PC_MAX_INPUT      3     // 最大输入长度。
46 #define PC_NAME_MAX      4     // 名称最大长度。
47 #define PC_PATH_MAX      5     // 路径最大长度。
48 #define PC_PIPE_BUF      6     // 管道缓冲大小。
49 #define PC_NO_TRUNC      7     // 文件名不截断。
50 #define PC_VDISABLE      8     // 禁止系统维护的终端特殊字符。
51 #define PC_CHOWN_RESTRICTED 9   // 不允许改变宿主。
52
53 #include <sys/stat.h> // 文件状态头文件。含有文件或文件系统状态结构 stat{} 和常量。
54 #include <sys/times.h> // 定义了进程中运行时间结构 tms 以及 times() 函数原型。
55 #include <sys/utsname.h> // 系统名称结构头文件。
56 #include <utime.h> // 用户时间头文件。定义了访问和修改时间结构以及 utime() 原型。
57
58 #ifdef __LIBRARY__
59
    // 以下是内核实现的系统调用符号常数，用作系统调用函数表中的索引值。（参见
    include/linux/sys.h）
60 #define __NR_setup      0       /* used only by init, to get system going */
    /* __NR_setup 仅用于初始化，以启动系统 */
61 #define __NR_exit      1
62 #define __NR_fork      2
63 #define __NR_read      3
64 #define __NR_write     4
65 #define __NR_open      5
66 #define __NR_close     6
67 #define __NR_waitpid   7
68 #define __NR_creat     8
69 #define __NR_link      9

```

70	<code>#define</code>	NR_unlink	10
71	<code>#define</code>	NR_execve	11
72	<code>#define</code>	NR_chdir	12
73	<code>#define</code>	NR_time	13
74	<code>#define</code>	NR_mknod	14
75	<code>#define</code>	NR_chmod	15
76	<code>#define</code>	NR_chown	16
77	<code>#define</code>	NR_break	17
78	<code>#define</code>	NR_stat	18
79	<code>#define</code>	NR_lseek	19
80	<code>#define</code>	NR_getpid	20
81	<code>#define</code>	NR_mount	21
82	<code>#define</code>	NR_umount	22
83	<code>#define</code>	NR_setuid	23
84	<code>#define</code>	NR_getuid	24
85	<code>#define</code>	NR_stime	25
86	<code>#define</code>	NR_ptrace	26
87	<code>#define</code>	NR_alarm	27
88	<code>#define</code>	NR_fstat	28
89	<code>#define</code>	NR_pause	29
90	<code>#define</code>	NR_utime	30
91	<code>#define</code>	NR_stty	31
92	<code>#define</code>	NR_gtty	32
93	<code>#define</code>	NR_access	33
94	<code>#define</code>	NR_nice	34
95	<code>#define</code>	NR_ftime	35
96	<code>#define</code>	NR_sync	36
97	<code>#define</code>	NR_kill	37
98	<code>#define</code>	NR_rename	38
99	<code>#define</code>	NR_mkdir	39
100	<code>#define</code>	NR_rmdir	40
101	<code>#define</code>	NR_dup	41
102	<code>#define</code>	NR_pipe	42
103	<code>#define</code>	NR_times	43
104	<code>#define</code>	NR_prof	44
105	<code>#define</code>	NR_brk	45
106	<code>#define</code>	NR_setgid	46
107	<code>#define</code>	NR_getgid	47
108	<code>#define</code>	NR_signal	48
109	<code>#define</code>	NR_geteuid	49
110	<code>#define</code>	NR_getegid	50
111	<code>#define</code>	NR_acct	51
112	<code>#define</code>	NR_phys	52
113	<code>#define</code>	NR_lock	53
114	<code>#define</code>	NR_ioctl	54
115	<code>#define</code>	NR_fcntl	55
116	<code>#define</code>	NR_mpx	56
117	<code>#define</code>	NR_setpgid	57
118	<code>#define</code>	NR_ulimit	58
119	<code>#define</code>	NR_uname	59
120	<code>#define</code>	NR_umask	60
121	<code>#define</code>	NR_chroot	61
122	<code>#define</code>	NR_ustat	62


```

123 #define __NR_dup2      63
124 #define __NR_getppid   64
125 #define __NR_getpgrp   65
126 #define __NR_setsid    66
127 #define __NR_sigaction 67
128 #define __NR_sgetmask  68
129 #define __NR_ssetmask  69
130 #define __NR_setreuid   70
131 #define __NR_setregid   71
132
    // 以下定义系统调用嵌入式汇编宏函数。
    // 不带参数的系统调用宏函数。type name(void)。
    // %0 - eax(__res), %1 - eax(__NR_###name)。其中 name 是系统调用的名称，与 __NR_ 组合形成上面
    // 的系统调用符号常数，从而用来对系统调用表中函数指针寻址。
    // 返回：如果返回值大于等于 0，则返回该值，否则置出错号 errno，并返回-1。
    // 在宏定义中，若在两个标记符号之间有两个连续的井号'##'，则表示在宏替换时会把这两个标记
    // 符号连接在一起。例如下面第 139 行上的__NR_###name，在替换了参数 name（例如是 fork）之后，
    // 最后在程序中出现的将会是符号__NR_fork。参见《The C Programming Language》附录 A.12.3。
133 #define __syscall0(type, name) \
134 type name(void) \
135 { \
136     long __res; \
137     __asm__ volatile ("int $0x80" \           // 调用系统中断 0x80。
138                       : "=a" (__res) \       // 返回值→eax(__res)。
139                       : "0" (__NR_###name)); \ // 输入为系统中断调用号__NR_name。
140     if (__res >= 0) \                         // 如果返回值>=0，则直接返回该值。
141         return (type) __res; \
142     errno = -__res; \                         // 否则置出错号，并返回-1。
143     return -1; \
144 }
145
    // 有 1 个参数的系统调用宏函数。type name(atype a)
    // %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a)。
146 #define __syscall1(type, name, atype, a) \
147 type name(atype a) \
148 { \
149     long __res; \
150     __asm__ volatile ("int $0x80" \
151                       : "=a" (__res) \
152                       : "0" (__NR_###name), "b" ((long) (a))); \
153     if (__res >= 0) \
154         return (type) __res; \
155     errno = -__res; \
156     return -1; \
157 }
158
    // 有 2 个参数的系统调用宏函数。type name(atype a, btype b)
    // %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b)。
159 #define __syscall2(type, name, atype, a, btype, b) \
160 type name(atype a, btype b) \
161 { \
162     long __res; \
163     __asm__ volatile ("int $0x80" \

```

```

164         : "=a" (__res) \
165         : "0" (__NR_##name), "b" ((long) (a)), "c" ((long) (b))); \
166 if (__res >= 0) \
167     return (type) __res; \
168 errno = -__res; \
169 return -1; \
170 }
171
// 有 3 个参数的系统调用宏函数。type name(atype a, btype b, ctype c)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b), %4 - edx(c)。
172 #define syscall3(type, name, atype, a, btype, b, ctype, c) \
173 type name(atype a, btype b, ctype c) \
174 { \
175     long __res; \
176     __asm__ volatile ("int $0x80" \
177         : "=a" (__res) \
178         : "0" (__NR_##name), "b" ((long) (a)), "c" ((long) (b)), "d" ((long) (c))); \
179 if (__res >= 0) \
180     return (type) __res; \
181 errno = -__res; \
182 return -1; \
183 }
184
185 #endif /* __LIBRARY__ */
186
187 extern int errno;                // 出错号, 全局变量。
188
// 对应各系统调用的函数原型定义。(详细说明参见 include/linux/sys.h)
189 int access(const char * filename, mode_t mode);
190 int acct(const char * filename);
191 int alarm(int sec);
192 int brk(void * end_data_segment);
193 void * sbrk(ptrdiff_t increment);
194 int chdir(const char * filename);
195 int chmod(const char * filename, mode_t mode);
196 int chown(const char * filename, uid_t owner, gid_t group);
197 int chroot(const char * filename);
198 int close(int fildes);
199 int creat(const char * filename, mode_t mode);
200 int dup(int fildes);
201 int execve(const char * filename, char ** argv, char ** envp);
202 int execv(const char * pathname, char ** argv);
203 int execvp(const char * file, char ** argv);
204 int execl(const char * pathname, char * arg0, ...);
205 int execlp(const char * file, char * arg0, ...);
206 int execle(const char * pathname, char * arg0, ...);
// 函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好一
// 些的代码, 更重要的是使用这个关键字可以避免产生某些 (未初始化变量的) 假警告信息。
// 等同于 gcc 的函数属性说明: void do_exit(int error_code) __attribute__((noreturn));
207 volatile void exit(int status);
208 volatile void _exit(int status);
209 int fcntl(int fildes, int cmd, ...);
210 int fork(void);

```

```

211 int getpid(void);
212 int getuid(void);
213 int geteuid(void);
214 int getgid(void);
215 int getegid(void);
216 int ioctl(int fildes, int cmd, ...);
217 int kill(pid\_t pid, int signal);
218 int link(const char * filename1, const char * filename2);
219 int lseek(int fildes, off\_t offset, int origin);
220 int mknod(const char * filename, mode\_t mode, dev\_t dev);
221 int mount(const char * specialfile, const char * dir, int rwflag);
222 int nice(int val);
223 int open(const char * filename, int flag, ...);
224 int pause(void);
225 int pipe(int * fildes);
226 int read(int fildes, char * buf, off\_t count);
227 int setpgrp(void);
228 int setpgid(pid\_t pid, pid\_t pgid);
229 int setuid(uid\_t uid);
230 int setgid(gid\_t gid);
231 void (*signal(int sig, void (*fn)(int)))(int);
232 int stat(const char * filename, struct stat * stat_buf);
233 int fstat(int fildes, struct stat * stat_buf);
234 int stime(time\_t * tptr);
235 int sync(void);
236 time\_t time(time\_t * tloc);
237 time\_t times(struct tms * tbuf);
238 int ulimit(int cmd, long limit);
239 mode\_t umask(mode\_t mask);
240 int umount(const char * specialfile);
241 int uname(struct utsname * name);
242 int unlink(const char * filename);
243 int ustat(dev\_t dev, struct ustat * ubuf);
244 int utime(const char * filename, struct utimbuf * times);
245 pid\_t waitpid(pid\_t pid, int * wait_stat, int options);
246 pid\_t wait(int * wait_stat);
247 int write(int fildes, const char * buf, off\_t count);
248 int dup2(int oldfd, int newfd);
249 int getppid(void);
250 pid\_t getpgrp(void);
251 pid\_t setsid(void);
252
253 #endif
254

```

14.14 utime.h 文件

14.14.1 功能描述

该文件定义了文件访问和修改时间结构 `utimbuf` 以及 `utime()` 函数原型。时间以秒计。





14.14.2 代码注释

程序 14-13 linux/include/utime.h

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

14.15 include/asm/目录下的文件

列表 14-2 linux/include/asm/目录下的文件

	名称	大小	最后修改时间 (GMT)	说明
	io.h	477 bytes	1991-08-07 10:17:51	m
	memory.h	507 bytes	1991-06-15 20:54:44	m
	segment.h	1366 bytes	1991-11-25 18:48:24	m
	system.h	1711 bytes	1991-09-17 13:08:31	m

14.16 io.h 文件

14.16.1 功能描述

该文件中定义了对硬件 IO 端口访问的嵌入式汇编宏函数：outb()、inb()以及 outb_p()和 inb_p()。前面两个函数与后面两个的主要区别在于后者代码中使用了 jmp 指令进行了时间延迟。

14.16.2 代码注释

程序 14-14 linux/include/asm/io.h

```

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

```

```

// 参数: port - 端口。返回读取的字节。
17 #define inb_p(port) ({ \
18 unsigned char _v; \
19 __asm__ volatile ("inb %%dx, %%al\n" \
20                  "\tjmp 1f\n" \
21                  "1:\tjmp 1f\n" \
22                  "1:": "=a" (_v): "d" (port)); \
23 _v; \
24 })
25

```

14.17 memory.h 文件

14.17.1 功能描述

该文件含有一个内存复制嵌入式汇编宏 `memcpy()`。与 `string.h` 中定义的 `memcpy()` 相同，只是后者采用的是嵌入式汇编 C 函数形式定义的。

14.17.2 代码注释

程序 14-15 linux/include/asm/memory.h

```

1  /*
2  *  NOTE!!! memcpy(dest,src,n) assumes ds=es=normal data segment. This
3  *  goes for all kernel functions (ds=es=kernel space, fs=local data,
4  *  gs=null), as well as for all well-behaving user programs (ds=es=
5  *  user data space). This is NOT a bug, as any user program that changes
6  *  es deserves to die if it isn't careful.
7  */
8  /*
9  *  注意!!!memcpy(dest,src,n)假设段寄存器 ds=es=通常数据段。在内核中使用的
10 *  所有函数都基于该假设 (ds=es=内核空间, fs=局部数据空间, gs=null), 具有良好
11 *  行为的应用程序也是这样 (ds=es=用户数据空间)。如果任何用户程序随意改动了
12 *  es 寄存器而出错, 则并不是由于系统程序错误造成的。
13 */
14 // 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。
15 // 参数: dest - 复制的目的地址, src - 复制的源地址, n - 复制字节数。
16 // %0 - edi(目的地址 dest), %1 - esi(源地址 src), %2 - ecx(字节数 n),
17 #define memcpy(dest,src,n) ({ \
18 void * _res = dest; \
19 __asm__ ("cld;rep;movsb" \
20          // 从 ds:[esi]复制到 es:[edi], 并且 esi++, edi++。
21          // 共复制 ecx(n)字节。
22          :: "D" ((long) (_res)), "S" ((long) (src)), "C" ((long) (n)) \
23          : "di", "si", "cx"); \
24 _res; \
25 })
26

```

14.18 segment.h 文件

14.18.1 功能描述

该文件中定义了一些访问 Intel CPU 中段寄存器或与段寄存器有关的内存操作函数。在 Linux 系统中，当用户程序通过系统调用开始执行内核代码时，内核程序会首先在段寄存器 `ds` 和 `es` 中加载全局描述符表 GDT 中的内核数据段描述符(段值 `0x10`)，即把 `ds` 和 `es` 用于访问内核数据段；而在 `fs` 中加载了局部描述符表 LDT 中的任务的数据段描述符(段值 `0x17`)，即把 `fs` 用于访问用户数据段。参见 `system_call.s` 第 89--93 行。因此在执行内核代码时，若要存取用户程序(任务)中的数据就需要使用特殊的方式。本文档中的 `get_fs_byte()` 和 `put_fs_byte()` 等函数就是专门用来访问用户程序中的数据。

14.18.2 代码注释

程序 14-16 linux/include/asm/segment.h

```

1  // 读取 fs 段中指定地址处的字节。
2  // 参数: addr - 指定的内存地址。
3  // %0 - (返回的字节_v); %1 - (内存地址 addr)。
4  // 返回: 返回内存 fs:[addr]处的字节。
5  // 第 3 行定义了一个寄存器变量_v，该变量将被保存在一个寄存器中，以便于高效访问和操作。
6  extern inline unsigned char get_fs_byte(const char * addr)
7  {
8      unsigned register char _v;
9
10     __asm__ ("movb %%fs:%1, %0": "=r" (_v): "m" (*addr));
11     return _v;
12 }
13
14 // 读取 fs 段中指定地址处的字。
15 // 参数: addr - 指定的内存地址。
16 // %0 - (返回的字_v); %1 - (内存地址 addr)。
17 // 返回: 返回内存 fs:[addr]处的字。
18 extern inline unsigned short get_fs_word(const unsigned short *addr)
19 {
20     unsigned short _v;
21
22     __asm__ ("movw %%fs:%1, %0": "=r" (_v): "m" (*addr));
23     return _v;
24 }
25
26 // 读取 fs 段中指定地址处的长字(4 字节)。
27 // 参数: addr - 指定的内存地址。
28 // %0 - (返回的长字_v); %1 - (内存地址 addr)。
29 // 返回: 返回内存 fs:[addr]处的长字。
30 extern inline unsigned long get_fs_long(const unsigned long *addr)
31 {
32     unsigned long _v;
33
34     __asm__ ("movl %%fs:%1, %0": "=r" (_v): "m" (*addr)); \
35     return _v;

```



```

23 }
24
    /// 将一字节存放在 fs 段中指定内存地址处。
    // 参数: val - 字节值; addr - 内存地址。
    // %0 - 寄存器(字节值 val); %1 - (内存地址 addr)。
25 extern inline void put\_fs\_byte(char val, char *addr)
26 {
27     __asm__ ("movb %0, %%fs:%1":: "r" (val), "m" (*addr));
28 }
29
    /// 将一字存放在 fs 段中指定内存地址处。
    // 参数: val - 字值; addr - 内存地址。
    // %0 - 寄存器(字值 val); %1 - (内存地址 addr)。
30 extern inline void put\_fs\_word(short val, short * addr)
31 {
32     __asm__ ("movw %0, %%fs:%1":: "r" (val), "m" (*addr));
33 }
34
    /// 将一长字存放在 fs 段中指定内存地址处。
    // 参数: val - 长字值; addr - 内存地址。
    // %0 - 寄存器(长字值 val); %1 - (内存地址 addr)。
35 extern inline void put\_fs\_long(unsigned long val, unsigned long * addr)
36 {
37     __asm__ ("movl %0, %%fs:%1":: "r" (val), "m" (*addr));
38 }
39
40 /*
41  * Someone who knows GNU asm better than I should double check the followig.
42  * It seems to work, but I don't know if I'm doing something subtly wrong.
43  * --- TYT, 11/24/91
44  * [ nothing wrong here, Linus ]
45  */
    /*
    * 比我更懂 GNU 汇编的人应该仔细检查下面的代码。这些代码能使用, 但我不知道是否
    * 含有一些小错误。
    * --- TYT, 1991 年 11 月 24 日
    * [ 这些代码没有错误, Linus ]
    */
46
    /// 取 fs 段寄存器值(选择符)。
    // 返回: fs 段寄存器值。
47 extern inline unsigned long get\_fs()
48 {
49     unsigned short _v;
50     __asm__ ("mov %%fs, %%ax":: "=a" (_v));
51     return _v;
52 }
53
    /// 取 ds 段寄存器值。
    // 返回: ds 段寄存器值。
54 extern inline unsigned long get\_ds()
55 {
56     unsigned short _v;

```

```

57     __asm__ ("mov %%ds, %%ax": "=a" (_v):);
58     return _v;
59 }
60
61     ///// 设置 fs 段寄存器。
62     // 参数: val - 段值 (选择符)。
63     extern inline void set_fs(unsigned long val)
64     {
65         __asm__ ("mov %0, %%fs": "=a" ((unsigned short) val));
66     }

```

14.19 system.h 文件

14.19.1 功能描述

该文件中定义了设置或修改描述符/中断门等的嵌入式汇编宏。其中，函数 `move_to_user_mode()` 是用于内核在初始化结束时人工切换（移动）到初始进程（任务 0）去执行，即从特权级 0 代码转移到特权级 3 的代码中去运行。所使用的方法是模拟中断调用返回过程，即利用 `iret` 指令来实现特权级的变更和堆栈的切换，从而把 CPU 执行控制流移动到初始任务 0 的环境中运行。见图 14-3 所示。

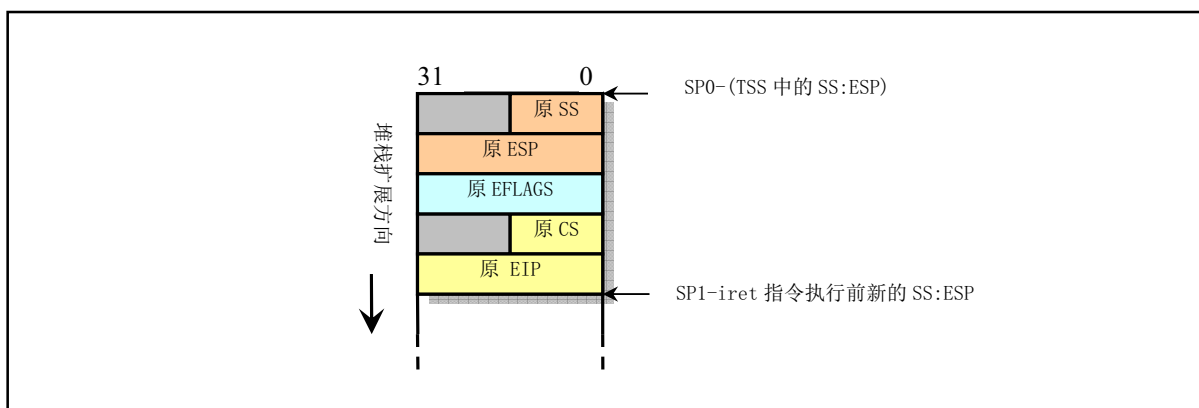


图 14-3 中断调用层间切换时堆栈内容

使用这种方法进行控制权的转移是由 CPU 保护机制造成的。CPU 允许低级别（例如特权级 3）的代码通过调用门或中断、陷阱门来调用或转移到高级别的代码中运行，但反之则不允许。因此内核采用了这种模拟 IRET 返回低级别代码的方法。

在去执行任务 0 代码之前，首先设置堆栈，模拟具有特权层切换的刚进入中断调用过程时堆栈的内容布置情况。然后执行 `iret` 指令，从而引起系统移到任务 0 中去执行。在执行 `iret` 语句时，堆栈内容如图 11.2 中所示，此时 `esp` 为 `esp1`。任务 0 的堆栈就是内核的堆栈。当执行了 `iret` 之后，就移到了任务 0 中执行了。由于任务 0 描述符特权级是 3，所以堆栈上的 `ss:esp` 也会被弹出。因此在 `iret` 之后，`esp` 又等于 `esp0` 了。注意，这里的中断返回指令 `iret` 并不会造成 CPU 去执行任务切换操作，因为在执行这个函数之前，标志位 `NT` 已经在 `sched_init()` 中被复位。在 `NT` 复位时执行 `iret` 指令不会造成 CPU 执行任务切换操作。任务 0 的执行纯粹是人工启动的。

任务 0 是一个特殊进程，它的数据段和代码段直接映射到内核代码和数据空间，即从物理地址 0 开始的 640K 内存空间，其堆栈地址即是内核代码所使用的堆栈。因此图中堆栈中的原 SS 和原 ESP 是将现有内核堆栈指针直接压入堆栈的。

该文件中的另一部份给出了在中断描述符表 IDT 中设置不同类型描述符项的宏。`_set_gate()`是一个多参数宏，它是设置中断门描述符宏 `set_intr_gate()`和设置陷阱门描述符宏 `set_trap_gate()`、`set_system_gate()`所调用的通用宏。IDT 表中的中断门（Interrupt Gate）和陷阱门（Trap Gate）描述符项的格式见图 14-4 所示。

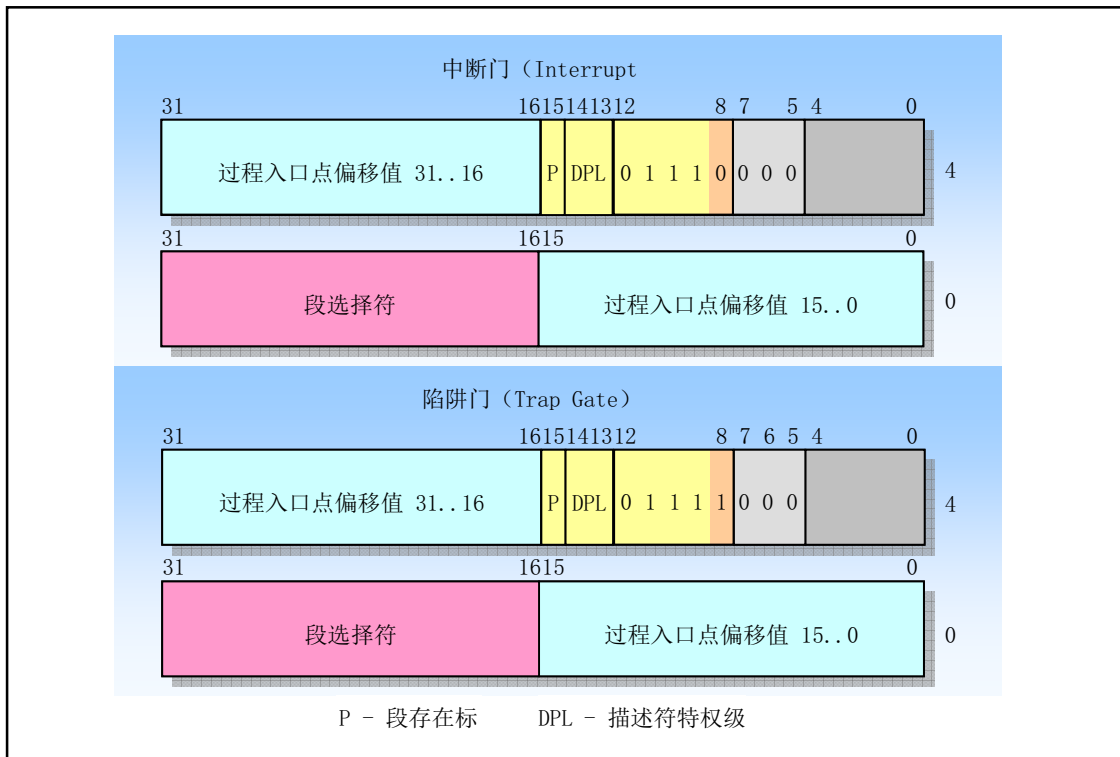


图 14-4 中断描述符表 IDT 中的中断门和陷阱门描述符格式

其中，P 是段存在标志；DPL 是描述符的优先级。中断门与陷阱门的区别在于对 EFLAGS 的中断允许标志 IF 的影响。通过中断门描述符执行的中断会复位 IF 标志，因此这种方式可以避免其它中断干扰当前中断的处理，并且随后的中断结束指令 IRET 会从堆栈上恢复 IF 标志的原值；而通过陷阱门执行的中断则不会影响 IF 标志。

在设置描述符的通用宏 `_set_gate(gate_addr, type, dpl, addr)` 中，参数 `gate_addr` 指定了描述符所处的物理内存地址。`type` 指明所需设置的描述符类型，它对应图 14-4 中描述符格式中第 6 字节的低 4 比特位，因此 `type=14 (0x0E)` 指明是中断门描述符，`type=15 (0x0F)` 指明是陷阱门描述符。参数 `dpl` 即对应描述符格式中的 DPL。`addr` 是描述符对应的中断处理过程的 32 位偏移地址。因为中断处理过程属于内核段代码，所以它们的段选择符值均为 0x0008（在 `eax` 寄存器高字中指定）。

`system.h` 文件的最后一部份是用于设置一般段描述符内容和在全局描述符表 GDT 中设置任务状态段描述符以及局部表段描述符的宏。这几个宏的参数含义与上述类似。

14.19.2 代码注释

程序 14-17 linux/include/asm/system.h

```

1111 移动用户模式运行。
1112 该函数利用 iret 指令实现从内核模式移动到初始任务 0 中去执行。
1113 #define move_to_user_mode() \
1114     __asm__ ( "movl %%esp, %%eax\n\t" \           // 保存堆栈指针 esp 到 eax 寄存器中。
1115             "pushl $0x17\n\t" \                 // 首先将堆栈段选择符(SS)入栈。
1116             "pushl %%eax\n\t" \                 // 然后将保存的堆栈指针值(esp)入栈。
1117             "pushfl\n\t" \                     // 将标志寄存器(eflags)内容入栈。
1118             "pushl $0x0f\n\t" \                 // 将 Task0 代码段选择符(cs)入栈。
1119             "pushl $1f\n\t" \                   // 将下面标号 1 的偏移地址(eip)入栈。
1120             "iret\n\t" \                       // 执行中断返回指令, 则会跳转到下面标号 1 处。
1121             "1: \tmovl $0x17, %%eax\n\t" \       // 此时开始执行任务 0,
1122             "movw %%ax, %%ds\n\t" \             // 初始化段寄存器指向本局部表的数据段。
1123             "movw %%ax, %%es\n\t" \
1124             "movw %%ax, %%fs\n\t" \
1125             "movw %%ax, %%gs" \
1126             ::: "ax" )
1127 #define sti() __asm__ ( "sti"::) // 开中断嵌入汇编宏函数。
1128 #define cli() __asm__ ( "cli"::) // 关中断。
1129 #define nop() __asm__ ( "nop"::) // 空操作。
1130 #define iret() __asm__ ( "iret"::) // 中断返回。
1131
1132 设置门描述符宏。
1133 根据参数中的中断或异常处理过程地址 addr、门描述符类型 type 和特权级信息 dpl, 设置位于
1134 地址 gate_addr 处的门描述符。(注意: 下面“偏移”值是相对于内核代码或数据段来说的)。
1135 参数: gate_addr - 描述符地址; type - 描述符类型域值; dpl - 描述符特权级; addr - 偏移地址。
1136 %0 - (由 dpl, type 组合成的类型标志字); %1 - (描述符低 4 字节地址);
1137 %2 - (描述符高 4 字节地址); %3 - edx(程序偏移地址 addr); %4 - eax(高字中含有段选择符 0x8)。
1138 #define set_gate(gate_addr, type, dpl, addr) \
1139     __asm__ ( "movw %%dx, %%ax\n\t" \           // 将偏移地址低字与选择符组合成描述符低 4 字节(eax)。
1140             "movw %0, %%dx\n\t" \             // 将类型标志字与偏移高字组合成描述符高 4 字节(edx)。
1141             "movl %%eax, %1\n\t" \             // 分别设置门描述符的低 4 字节和高 4 字节。
1142             "movl %%edx, %2" \
1143             : \
1144             : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
1145             "o" (*((char *) (gate_addr))), \
1146             "o" (*(4+(char *) (gate_addr))), \
1147             "d" ((char *) (addr)), "a" (0x00080000)
1148
1149 设置中断门函数(自动屏蔽随后的中断)。
1150 参数: n - 中断号; addr - 中断程序偏移地址。
1151 &idt[n] 是中断描述符表中中断号 n 对应项的偏移值; 中断描述符的类型是 14, 特权级是 0。
1152 #define set_intr_gate(n, addr) \
1153     set_gate(&idt[n], 14, 0, addr)
1154
1155 设置陷阱门函数。
1156 参数: n - 中断号; addr - 中断程序偏移地址。
1157 &idt[n] 是中断描述符表中中断号 n 对应项的偏移值; 中断描述符的类型是 15, 特权级是 0。
1158 #define set_trap_gate(n, addr) \
1159     set_gate(&idt[n], 15, 0, addr)
1160
1161 设置系统陷阱门函数。

```











```

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

```

14.20 include/linux/目录下的文件

列表 14-3 linux/include/linux/目录下的文件

名称	大小	最后修改时间 (GMT)	说明
 config.h	1289 bytes	1991-12-08 18:37:16	m
 fdreg.h	2466 bytes	1991-11-02 10:48:44	m
 fs.h	5474 bytes	1991-12-01 19:48:26	m
 hdreg.h	1968 bytes	1991-10-13 15:32:15	m
 head.h	304 bytes	1991-06-19 19:24:13	m
 kernel.h	734 bytes	1991-12-02 03:19:07	m
 mm.h	219 bytes	1991-07-29 17:51:12	m
 sched.h	5838 bytes	1991-11-20 14:40:46	m
 sys.h	2588 bytes	1991-11-25 20:15:35	m
 tty.h	2173 bytes	1991-09-21 11:58:05	m

14.21 config.h 文件

14.21.1 功能描述

内核配置头文件。定义使用的键盘语言类型和硬盘类型（HD_TYPE）可选项。

14.21.2 代码注释

程序 14-18 linux/include/linux/config.h

```

1 #ifndef CONFIG\_H
2 #define CONFIG\_H
3
4 /*
5  * The root-device is no longer hard-coded. You can change the default
6  * root-device by changing the line ROOT_DEV = XXX in boot/bootsect.s
7  */
8 /*
9  * 根文件系统设备已不再是硬编码的了。通过修改 boot/bootsect.s 文件中行
10  * ROOT_DEV = XXX，你可以改变根设备的默认设置值。
11  */
12
13 /*
14  * define your keyboard here -
15  * KBD_FINNISH for Finnish keyboards

```

```

12 * KBD_US for US-type
13 * KBD_GR for German keyboards
14 * KBD_FR for Frech keyboard
15 */
/*
* 在这里定义你的键盘类型 -
* KBD_FINNISH 是芬兰键盘。
* KBD_US 是美式键盘。
* KBD_GR 是德式键盘。
* KBD_FR 是法式键盘。
*/
16 /**define KBD_US */
17 /**define KBD_GR */
18 /**define KBD_FR */
19 #define KBD_FINNISH
20
21 /*
22 * Normally, Linux can get the drive parameters from the BIOS at
23 * startup, but if this for some unfathomable reason fails, you'd
24 * be left stranded. For this case, you can define HD_TYPE, which
25 * contains all necessary info on your harddisk.
26 *
27 * The HD_TYPE macro should look like this:
28 *
29 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
30 *
31 * In case of two harddisks, the info should be sepatated by
32 * commas:
33 *
34 * #define HD_TYPE { h, s, c, wpcom, lz, ctl }, { h, s, c, wpcom, lz, ctl }
35 */
/*
* 通常, Linux 能够在启动时从 BIOS 中获取驱动器德参数, 但是若由于未知原因
* 而没有得到这些参数时, 会使程序束手无策。对于这种情况, 你可以定义 HD_TYPE,
* 其中包括硬盘的所有信息。
*
* HD_TYPE 宏应该象下面这样的形式:
*
* #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
*
* 对于有两个硬盘的情况, 参数信息需用逗号分开:
*
* #define HD_TYPE { h, s, c, wpcom, lz, ctl }, {h, s, c, wpcom, lz, ctl }
*/
36 /*
37 This is an example, two drives, first is type 2, second is type 3:
38
39 #define HD_TYPE { 4, 17, 615, 300, 615, 8 }, { 6, 17, 615, 300, 615, 0 }
40
41 NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
42 with more than 8 heads.
43
44 If you want the BIOS to tell what kind of drive you have, just

```

```

45  leave HD_TYPE undefined. This is the normal thing to do.
46  */
/*
 * 下面是一个例子，两个硬盘，第 1 个是类型 2，第 2 个是类型 3：
 *
 * #define HD_TYPE { 4, 17, 615, 300, 615, 8 }, {6, 17, 615, 300, 615, 0 }
 *
 * 注意：对应所有硬盘，若其磁头数<=8，则 ctl 等于 0，若磁头数多于 8 个，
 * 则 ctl=8。
 *
 * 如果你想让 BIOS 给出硬盘的类型，那么只需不定义 HD_TYPE。这是默认操作。
 */
47
48 #endif
49

```

14.22 fdreg.h 头文件

14.22.1 功能描述

该头文件用以说明软盘系统常用到的一些参数以及所使用的 I/O 端口。由于软盘驱动器的控制比较烦琐，命令也多，因此在阅读代码之前，最好先参考有关微型计算机控制接口原理的书籍，了解软盘控制器(FDC)的工作原理，然后你就会觉得这里的定义还是比较合理有序的。

在编程时需要访问 4 个端口，分别对应一个或多个寄存器。对于 1.2M 的软盘控制器有表 14-1 中一些端口。

表 14-1 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。

14.22.2 文件注释

程序 14-19 linux/include/linux/fdreg.h

```

1 /*
2  * This file contains some defines for the floppy disk controller.
3  * Various sources. Mostly "IBM Microcomputers: A Programmers
4  * Handbook", Sanches and Canton.
5  */
6 /*
7  * 该文件中含有一些软盘控制器的一些定义。这些信息有多处来源，大多数取自 Sanches 和 Canton
8  * 编著的"IBM 微型计算机：程序员手册"一书。
9  */
10 #ifndef FDREG_H // 该定义用来排除代码中重复包含此头文件。
11 #define FDREG_H
12
13 // 一些软盘类型函数的原型说明。
14 extern int ticks to floppy on(unsigned int nr);
15 extern void floppy on(unsigned int nr);
16 extern void floppy off(unsigned int nr);
17 extern void floppy select(unsigned int nr);
18 extern void floppy deselect(unsigned int nr);
19
20 // 下面是有关软盘控制器一些端口和符号的定义。
21 /* Fd controller regs. S&C, about page 340 */
22 /* 软盘控制器 (FDC) 寄存器端口。摘自 S&C 书中约 340 页 */
23 #define FD STATUS 0x3f4 // 主状态寄存器端口。
24 #define FD DATA 0x3f5 // 数据端口。
25 #define FD DOR 0x3f2 /* Digital Output Register */
26 // 数字输出寄存器（也称为数字控制寄存器）。
27 #define FD DIR 0x3f7 /* Digital Input Register (read) */
28 // 数字输入寄存器。
29 #define FD DCR 0x3f7 /* Diskette Control Register (write) */
30 // 数据传输率控制寄存器。
31
32 /* Bits of main status register */
33 /* 主状态寄存器各比特位的含义 */
34 #define STATUS BUSYMASK 0x0F /* drive busy mask */
35 // 驱动器忙位（每位对应一个驱动器）。
36 #define STATUS_BUSY 0x10 /* FDC busy */
37 // 软盘控制器忙。

```

```

25 #define STATUS_DMA      0x20      /* 0- DMA mode */
                                   // 0 - 为 DMA 数据传输模式, 1 - 为非 DMA 模式。
26 #define STATUS_DIR      0x40      /* 0- cpu->fdc */
                                   // 传输方向: 0 - CPU → fdc, 1 - 相反。
27 #define STATUS_READY    0x80      /* Data reg ready */
                                   // 数据寄存器就绪位。
28
29 /* Bits of FD_ST0 */
   /*状态字节 0 (ST0) 各比特位的含义 */
30 #define ST0_DS          0x03      /* drive select mask */
                                   // 驱动器选择号 (发生中断时驱动器号)。
31 #define ST0_HA          0x04      /* Head (Address) */
                                   // 磁头号。
32 #define ST0_NR          0x08      /* Not Ready */
                                   // 磁盘驱动器未准备好。
33 #define ST0_ECE         0x10      /* Equipment chech error */
                                   // 设备检测出错 (零磁道校准出错)。
34 #define ST0_SE          0x20      /* Seek end */
                                   // 寻道或重新校正操作执行结束。
35 #define ST0_INTR        0xC0      /* Interrupt code mask */
                                   // 中断代码位 (中断原因), 00 - 命令正常结束;
                                   // 01 - 命令异常结束; 10 - 命令无效; 11 - FDD 就绪状态改变。
36
37 /* Bits of FD_ST1 */
   /*状态字节 1 (ST1) 各比特位的含义 */
38 #define ST1_MAM         0x01      /* Missing Address Mark */
                                   // 未找到地址标志 (ID AM)。
39 #define ST1_WP          0x02      /* Write Protect */
                                   // 写保护。
40 #define ST1_ND          0x04      /* No Data - unreadable */
                                   // 未找到指定的扇区。
41 #define ST1_OR          0x10      /* OverRun */
                                   // 数据传输超时 (DMA 控制器故障)。
42 #define ST1_CRC         0x20      /* CRC error in data or addr */
                                   // CRC 检验出错。
43 #define ST1_EOC         0x80      /* End Of Cylinder */
                                   // 访问超过一个磁道上的最大扇区号。
44
45 /* Bits of FD_ST2 */
   /*状态字节 2 (ST2) 各比特位的含义 */
46 #define ST2_MAM         0x01      /* Missing Address Mark (again) */
                                   // 未找到数据地址标志。
47 #define ST2_BC          0x02      /* Bad Cylinder */
                                   // 磁道坏。
48 #define ST2_SNS         0x04      /* Scan Not Satisfied */
                                   // 检索 (扫描) 条件不满足。
49 #define ST2_SEH         0x08      /* Scan Equal Hit */
                                   // 检索条件满足。
50 #define ST2_WC          0x10      /* Wrong Cylinder */
                                   // 磁道 (柱面) 号不符。
51 #define ST2_CRC         0x20      /* CRC error in data field */
                                   // 数据场 CRC 校验错。
52 #define ST2_CM          0x40      /* Control Mark = deleted */

```

```

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

```

14.23 fs.h 文件

14.23.1 功能描述

fs.h 头文件中定义了有关文件系统的一些常数和结构。主要包含高速缓冲区中缓冲块的数据结构、MINIX 1.0 文件系统中超级块和 i 节点结构以及文件表结构和一些管道操作宏。

14.23.2 代码注释

程序 14-20 linux/include/linux/fs.h

```

1 /*
2  * This file has definitions for some important file table
3  * structures etc.
4  */
/*

```

```

* 本文件含有某些重要文件表结构的定义等。
*/
5
6 #ifndef FS_H
7 #define FS_H
8
9 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
10
11 /* devices are as follows: (same as minix, so we can use the minix
12  * file system. These are major numbers.)
13  *
14  * 0 - unused (nodev)
15  * 1 - /dev/mem
16  * 2 - /dev/fd
17  * 3 - /dev/hd
18  * 4 - /dev/ttyx
19  * 5 - /dev/tty
20  * 6 - /dev/lp
21  * 7 - unnamed pipes
22  */
/*
* 系统所含的设备如下：（与 minix 系统的一样，所以我们可以使用 minix 的
* 文件系统。以下这些是主设备号。）
*
* 0 - 没有用到（nodev）
* 1 - /dev/mem      内存设备。
* 2 - /dev/fd       软盘设备。
* 3 - /dev/hd       硬盘设备。
* 4 - /dev/ttyx     tty 串行终端设备。
* 5 - /dev/tty      tty 终端设备。
* 6 - /dev/lp       打印设备。
* 7 - unnamed pipes 没有命名的管道。
*/
23
24 #define IS_SEEKABLE(x) ((x)>=1 && (x)<=3)    // 判断设备是否是可寻找定位的。
25
26 #define READ 0
27 #define WRITE 1
28 #define READA 2    /* read-ahead - don't pause */
29 #define WRITEA 3    /* "write-ahead" - silly, but somewhat useful */
30
31 void buffer_init(long buffer_end);
32
33 #define MAJOR(a) (((unsigned)(a))>>8)    // 取高字节（主设备号）。
34 #define MINOR(a) ((a)&0xff)    // 取低字节（次设备号）。
35
36 #define NAME_LEN 14    // 名字长度值。
37 #define ROOT_INO 1    // 根 i 节点。
38
39 #define I_MAP_SLOTS 8    // i 节点位图槽数。
40 #define Z_MAP_SLOTS 8    // 逻辑块（区段块）位图槽数。
41 #define SUPER_MAGIC 0x137F    // 文件系统魔数。
42

```

```

43 #define NR_OPEN 20 // 打开文件数。
44 #define NR_INODE 32 // 系统同时最多使用 I 节点个数。
45 #define NR_FILE 64 // 系统最多文件个数（文件数组项数）。
46 #define NR_SUPER 8 // 系统所含超级块个数（超级块数组项数）。
47 #define NR_HASH 307 // 缓冲区 Hash 表数组项数值。
48 #define NR_BUFFERS nr_buffers // 系统所含缓冲块个数。初始化后不再改变。
49 #define BLOCK_SIZE 1024 // 数据块长度（字节值）。
50 #define BLOCK_SIZE_BITS 10 // 数据块长度所占比特位数。
51 #ifndef NULL
52 #define NULL ((void *) 0)
53 #endif
54
// 每个逻辑块可存放的 i 节点数。
55 #define INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct d_inode)))
// 每个逻辑块可存放的目录项数。
56 #define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))
57
// 管道头、管道尾、管道大小、管道空?、管道满?、管道头指针递增。
58 #define PIPE_HEAD(inode) ((inode).i_zone[0])
59 #define PIPE_TAIL(inode) ((inode).i_zone[1])
60 #define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
61 #define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
62 #define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))
63 #define INC_PIPE(head) \
64 __asm__ ("incl %0|n|tandl $4095,%0"::"m" (head))
65
66 typedef char buffer_block[BLOCK_SIZE]; // 块缓冲区。
67
// 缓冲块头数据结构。（极为重要!!!）
// 在程序中常用 bh 来表示 buffer_head 类型的缩写。
68 struct buffer_head {
69     char * b_data; // /* pointer to data block (1024 bytes) */ // 指针。
70     unsigned long b_blocknr; // /* block number */ // 块号。
71     unsigned short b_dev; // /* device (0 = free) */ // 数据源的设备号。
72     unsigned char b_uptodate; // 更新标志：表示数据是否已更新。
73     unsigned char b_dirty; // /* 0-clean, 1-dirty */ // 修改标志：0 未修改，1 已修改。
74     unsigned char b_count; // /* users using this block */ // 使用的用户数。
75     unsigned char b_lock; // /* 0 - ok, 1 -locked */ // 缓冲区是否被锁定。
76     struct task_struct * b_wait; // 指向等待该缓冲区解锁的任务。
77     struct buffer_head * b_prev; // hash 队列上一块（这四个指针用于缓冲区的管理）。
78     struct buffer_head * b_next; // hash 队列下一块。
79     struct buffer_head * b_prev_free; // 空闲表上一块。
80     struct buffer_head * b_next_free; // 空闲表下一块。
81 };
82
// 磁盘上的索引节点(i 节点)数据结构。
83 struct d_inode {
84     unsigned short i_mode; // 文件类型和属性(rwx 位)。
85     unsigned short i_uid; // 用户 id (文件拥有者标识符)。
86     unsigned long i_size; // 文件大小（字节数）。
87     unsigned long i_time; // 修改时间（自 1970.1.1:0 算起，秒）。
88     unsigned char i_gid; // 组 id (文件拥有者所在的组)。
89     unsigned char i_nlinks; // 链接数（多少个文件目录项指向该 i 节点）。

```

```

90     unsigned short i_zone[9];    // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
                                   // zone 是区的意思，可译成区段，或逻辑块。
91 };
92
93 // 这是在内存中的 i 节点结构。前 7 项与 d_inode 完全一样。
94 struct m_inode {
95     unsigned short i_mode;        // 文件类型和属性(rwx 位)。
96     unsigned short i_uid;        // 用户 id (文件拥有者标识符)。
97     unsigned long i_size;        // 文件大小 (字节数)。
98     unsigned long i_mtime;       // 修改时间 (自 1970.1.1:0 算起，秒)。
99     unsigned char i_gid;        // 组 id (文件拥有者所在的组)。
100    unsigned char i_nlinks;       // 文件目录项链接数。
101    unsigned short i_zone[9];     // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
102    /* these are in memory also */
103    struct task_struct * i_wait;   // 等待该 i 节点的进程。
104    unsigned long i_atime;        // 最后访问时间。
105    unsigned long i_ctime;        // i 节点自身修改时间。
106    unsigned short i_dev;        // i 节点所在的设备号。
107    unsigned short i_num;        // i 节点号。
108    unsigned short i_count;       // i 节点被使用的次数，0 表示该 i 节点空闲。
109    unsigned char i_lock;        // 锁定标志。
110    unsigned char i_dirt;        // 已修改(脏)标志。
111    unsigned char i_pipe;        // 管道标志。
112    unsigned char i_mount;       // 安装标志。
113    unsigned char i_seek;        // 搜寻标志(lseek 时)。
114    unsigned char i_update;      // 更新标志。
115 };
116
117 // 文件结构 (用于在文件句柄与 i 节点之间建立关系)
118 struct file {
119     unsigned short f_mode;       // 文件操作模式 (RW 位)
120     unsigned short f_flags;      // 文件打开和控制的标志。
121     unsigned short f_count;      // 对应文件引用计数值。
122     struct m_inode * f_inode;    // 指向对应 i 节点。
123     off_t f_pos;                // 文件位置 (读写偏移值)。
124 };
125
126 // 内存中磁盘超级块结构。
127 struct super_block {
128     unsigned short s_ninodes;    // 节点数。
129     unsigned short s_nzones;    // 逻辑块数。
130     unsigned short s_imap_blocks; // i 节点位图所占用的数据块数。
131     unsigned short s_zmap_blocks; // 逻辑块位图所占用的数据块数。
132     unsigned short s_firstdatazone; // 第一个数据逻辑块号。
133     unsigned short s_log_zone_size; // log(数据块数/逻辑块)。(以 2 为底)。
134     unsigned long s_max_size;    // 文件最大长度。
135     unsigned short s_magic;      // 文件系统魔数。
136     /* These are only in memory */
137     struct buffer_head * s_imap[8]; // i 节点位图缓冲块指针数组 (占用 8 块，可表示 64M)。
138     struct buffer_head * s_zmap[8]; // 逻辑块位图缓冲块指针数组 (占用 8 块)。
139     unsigned short s_dev;        // 超级块所在的设备号。
140     struct m_inode * s_isup;     // 被安装的文件系统根目录的 i 节点。(isup=super i)
141     struct m_inode * s_imount;   // 被安装到的 i 节点。

```



```

139     unsigned long s_time;           // 修改时间。
140     struct task\_struct * s_wait;   // 等待该超级块的进程。
141     unsigned char s_lock;           // 被锁定标志。
142     unsigned char s_rd_only;        // 只读标志。
143     unsigned char s_dirt;           // 已修改(脏)标志。
144 };
145
146 // 磁盘上超级块结构。上面 125-132 行完全一样。
147 struct d\_super\_block {
148     unsigned short s_ninodes;       // 节点数。
149     unsigned short s_nzones;        // 逻辑块数。
150     unsigned short s_imap_blocks;    // i 节点位图所占用的数据块数。
151     unsigned short s_zmap_blocks;    // 逻辑块位图所占用的数据块数。
152     unsigned short s_firstdatazone; // 第一个数据逻辑块。
153     unsigned short s_log_zone_size; // log(数据块数/逻辑块)。(以 2 为底)。
154     unsigned long s_max_size;        // 文件最大长度。
155     unsigned short s_magic;          // 文件系统魔数。
156 };
157
158 // 文件目录项结构。
159 struct dir\_entry {
160     unsigned short inode;            // i 节点号。
161     char name[NAME_LEN];            // 文件名, 长度 NAME_LEN=14。
162 };
163
164 extern struct m\_inode inode\_table[NR_INODE]; // 定义 i 节点表数组 (32 项)。
165 extern struct file file\_table[NR_FILE];      // 文件表数组 (64 项)。
166 extern struct super\_block super\_block[NR_SUPER]; // 超级块数组 (8 项)。
167 extern struct buffer\_head * start\_buffer;    // 缓冲区起始内存位置。
168 extern int nr\_buffers;                      // 缓冲块数。
169
170 // 磁盘操作函数原型。
171 // 检测驱动器中软盘是否改变。
172 extern void check\_disk\_change(int dev);
173 // 检测指定软驱中软盘更换情况。如果软盘更换了则返回 1, 否则返回 0。
174 extern int floppy\_change(unsigned int nr);
175 // 设置启动指定驱动器所需等待的时间 (设置等待定时器)。
176 extern int ticks\_to\_floppy\_on(unsigned int dev);
177 // 启动指定驱动器。
178 extern void floppy\_on(unsigned int dev);
179 // 关闭指定的软盘驱动器。
180 extern void floppy\_off(unsigned int dev);
181
182 // 以下是文件系统操作管理用的函数原型。
183 // 将 i 节点指定的文件截为 0。
184 extern void truncate(struct m\_inode * inode);
185 // 刷新 i 节点信息。
186 extern void sync\_inodes(void);
187 // 等待指定的 i 节点。
188 extern void wait\_on(struct m\_inode * inode);
189 // 逻辑块(区段, 磁盘块)位图操作。取数据块 block 在设备上对应的逻辑块号。
190 extern int bmap(struct m\_inode * inode, int block);
191 // 创建数据块 block 在设备上对应的逻辑块, 并返回在设备上的逻辑块号。

```

```

177 extern int create\_block(struct m\_inode * inode, int block);
    // 获取指定路径名的 i 节点号。
178 extern struct m\_inode * namei(const char * pathname);
    // 根据路径名为打开文件操作作准备。
179 extern int open\_namei(const char * pathname, int flag, int mode,
180     struct m\_inode ** res_inode);
    // 释放一个 i 节点(回写入设备)。
181 extern void iput(struct m\_inode * inode);
    // 从设备读取指定节点号的一个 i 节点。
182 extern struct m\_inode * iget(int dev, int nr);
    // 从 i 节点表(inode_table)中获取一个空闲 i 节点项。
183 extern struct m\_inode * get\_empty\_inode(void);
    // 获取(申请一)管道节点。返回为 i 节点指针(如果是 NULL 则失败)。
184 extern struct m\_inode * get\_pipe\_inode(void);
    // 在哈希表中查找指定的数据块。返回找到块的缓冲头指针。
185 extern struct buffer\_head * get\_hash\_table(int dev, int block);
    // 从设备读取指定块(首先会在 hash 表中查找)。
186 extern struct buffer\_head * getblk(int dev, int block);
    // 读/写数据块。
187 extern void ll\_rw\_block(int rw, struct buffer\_head * bh);
    // 释放指定缓冲块。
188 extern void brelse(struct buffer\_head * buf);
    // 读取指定的数据块。
189 extern struct buffer\_head * bread(int dev, int block);
    // 读 4 块缓冲区到指定地址的内存中。
190 extern void bread\_page(unsigned long addr, int dev, int b[4]);
    // 读取头一个指定的数据块, 并标记后续将要读的块。
191 extern struct buffer\_head * breada(int dev, int block, ...);
    // 向设备 dev 申请一个磁盘块(区段, 逻辑块)。返回逻辑块号
192 extern int new\_block(int dev);
    // 释放设备数据区中的逻辑块(区段, 磁盘块)block。复位指定逻辑块 block 的逻辑块位图比特位。
193 extern void free\_block(int dev, int block);
    // 为设备 dev 建立一个新 i 节点, 返回 i 节点号。
194 extern struct m\_inode * new\_inode(int dev);
    // 释放一个 i 节点(删除文件时)。
195 extern void free\_inode(struct m\_inode * inode);
    // 刷新指定设备缓冲区。
196 extern int sync\_dev(int dev);
    // 读取指定设备的超级块。
197 extern struct super\_block * get\_super(int dev);
198 extern int ROOT\_DEV;
199
    // 安装根文件系统。
200 extern void mount\_root(void);
201
202 #endif
203

```

14.24 hdreg.h 文件

14.24.1 功能描述

该文件中主要定义了对硬盘控制器进行编程的一些命令常量符号。其中包括控制器端口、硬盘状态寄存器各位的状态、控制器命令以及出错状态常量符号。另外还给出了硬盘分区表数据结构。

14.24.2 代码注释

程序 14-21 linux/include/linux/hdreg.h

```

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

```

```

35 #define WIN_READ          0x20    // 读扇区。
36 #define WIN_WRITE        0x30    // 写扇区。
37 #define WIN_VERIFY       0x40    // 扇区检验。
38 #define WIN_FORMAT       0x50    // 格式化磁道。
39 #define WIN_INIT         0x60    // 控制器初始化。
40 #define WIN_SEEK         0x70    // 寻道。
41 #define WIN_DIAGNOSE     0x90    // 控制器诊断。
42 #define WIN_SPECIFY      0x91    // 建立驱动器参数。
43
44 /* Bits for HD_ERROR */
45 /* 错误寄存器各比特位的含义 (HD_ERROR) */
46 // 执行控制器诊断命令时含义与其他命令时的不同。下面分别列出：
47 // =====
48 //          诊断命令时          其他命令时
49 // -----
50 // 0x01      无错误              数据标志丢失
51 // 0x02      控制器出错          磁道 0 错
52 // 0x03      扇区缓冲区错
53 // 0x04      ECC 部件错          命令放弃
54 // 0x05      控制处理器错
55 // 0x10              ID 未找到
56 // 0x40              ECC 错误
57 // 0x80              坏扇区
58 //-----
59 #define MARK_ERR          0x01    /* Bad address mark ? */
60 #define TRKO_ERR          0x02    /* couldn't find track 0 */
61 #define ABRT_ERR          0x04    /* ? */
62 #define ID_ERR            0x10    /* ? */
63 #define ECC_ERR           0x40    /* ? */
64 #define BBD_ERR           0x80    /* ? */
65
66 // 硬盘分区表结构。参见下面列表后信息。
67 struct partition {
68     unsigned char boot_ind;        /* 0x80 - active (unused) */
69     unsigned char head;            /* ? */
70     unsigned char sector;          /* ? */
71     unsigned char cyl;             /* ? */
72     unsigned char sys_ind;         /* ? */
73     unsigned char end_head;        /* ? */
74     unsigned char end_sector;      /* ? */
75     unsigned char end_cyl;         /* ? */
76     unsigned int start_sect;       /* starting sector counting from 0 */
77     unsigned int nr_sects;         /* nr of sectors in partition */
78 };
79 #endif

```

14.24.3 其他信息

14.24.3.1 硬盘分区表

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1--4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小

和起止的柱面号、磁道号和扇区号，见表 14-2 所示。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE--0x1FD 处。

表 14 - 2 硬盘分区表结构

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。
0x02	sector	字节	分区起始扇区号(位 0-5)和起始柱面号高 2 位(位 6-7)。
0x03	cyl	字节	分区起始柱面号低 8 位。
0x04	sys_ind	字节	分区类型字节。0x0b-DOS; 0x80-Old Minix; 0x83-Linux ...
0x05	end_head	字节	分区的结束磁头号。
0x06	end_sector	字节	结束扇区号(位 0-5)和结束柱面号高 2 位(位 6-7)。
0x07	end_cyl	字节	结束柱面号低 8 位。
0x08--0x0b	start_sect	长字	分区起始物理扇区号。
0x0c--0x0f	nr_sects	长字	分区占用的扇区数。

14.25 head.h 文件

14.25.1 功能描述

head 头文件，定义了 Intel CPU 中描述符的简单结构，和指定描述符的项号。

14.25.2 代码注释

程序 14-22 linux/include/linux/head.h

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

```
20 #endif
21
```

14.26 kernel.h 文件

14.26.1 功能描述

定义了一些内核常用的函数原型等。

14.26.2 代码注释

程序 14-23 linux/include/linux/kernel.h

```
1 /*
2  * 'kernel.h' contains some often-used function prototypes etc
3  */
4 /*
5  * 'kernel.h' 定义了一些常用函数的原型等。
6  */
7 // 验证给定地址开始的内存块是否超限。若超限则追加内存。( kernel/fork.c, 24 )。
8 void verify\_area(void * addr, int count);
9 // 显示内核出错信息，然后进入死循环。( kernel/panic.c, 16 )。
10 // 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
11 // 一些的代码，更重要的是使用这个关键字可以避免产生某未初始化变量的假警告信息。
12 volatile void panic(const char * str);
13 // 标准打印(显示)函数。( init/main.c, 151)。
14 int printf(const char * fmt, ...);
15 // 内核专用的打印信息函数，功能与 printf() 相同。( kernel/printk.c, 21 )。
16 int printk(const char * fmt, ...);
17 // 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
18 int tty\_write(unsigned ch, char * buf, int count);
19 // 通用内核内存分配函数。( lib/malloc.c, 117)。
20 void * malloc(unsigned int size);
21 // 释放指定对象占用的内存。( lib/malloc.c, 182)。
22 void free\_s(void * obj, int size);
23
24 #define free(x) free\_s((x), 0)
25
26 /*
27  * This is defined as a macro, but at some point this might become a
28  * real subroutine that sets a flag if it returns true (to do
29  * BSD-style accounting where the process is flagged if it uses root
30  * privs). The implication of this is that you should do normal
31  * permissions checks first, and check suser() last.
32  */
33 /*
34  * 下面函数是以宏的形式定义的，但是在某方面来看它可以成为一个真正的子程序，
35  * 如果返回是 true 时它将设置标志（如果使用 root 用户权限的进程设置了标志，则用
36  * 于执行 BSD 方式的计帐处理）。这意味着你应该首先执行常规权限检查，最后再
37  * 检测 suser()。
```

```

    */
21 #define suser() (current->euid == 0)           // 检测是否是超级用户。
22
23

```

14.27 mm.h 文件

14.27.1 功能描述

mm.h 是内存管理头文件。其中主要定义了内存页面的大小和几个页面释放函数原型。

14.27.2 代码注释

程序 14-24 linux/include/linux/mm.h

```

1 #ifndef MM_H
2 #define MM_H
3
4 #define PAGE_SIZE 4096           // 定义内存页面的大小(字节数)。
5
6 // 取空闲页面函数。返回页面地址。扫描页面映射数组 mem_map[]取空闲页面。
7 extern unsigned long get_free_page(void);
8 // 在指定线性地址处映射一内存页面。在页目录和页表中设置该页面信息。返回该页面物理地址。
9 extern unsigned long put_page(unsigned long page, unsigned long address);
10 // 释放物理地址 addr 开始的一页面内存。修改页面映射数组 mem_map[]中引用次数信息。
11 extern void free_page(unsigned long addr);
12
13 #endif
14

```

14.28 sched.h 文件

14.28.1 功能描述

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以及任务上下文切换 `switch_to()` 的嵌入式汇编函数宏。下面详细描述一下任务切换宏的执行过程。

任务切换宏 `switch_to(n)` (从 171 行开始) 首先声明了一个结构 `'struct {long a,b;} __tmp'`，用于在任务内核态堆栈上保留出 8 字节的空间来存放将切换到新任务的任务状态段 TSS 的选择符。然后测试我们是否是在执行切换到当前任务的操作，如果是则什么也不需要，直接退出。否则就把新任务 TSS 的选择符保存到临时结构 `__tmp` 中的偏移位置 4 处，此时 `__tmp` 中的数据设置为：

```

__tmp+0: 未定义 (long)
__tmp+4: 新任务 TSS 的选择符 (word)
__tmp+6: 未定义 (word)

```

接下来把 `%ecx` 寄存器中的新任务指针与全局变量 `current` 中的当前任务指针相交换，让 `current` 含有我们将要切换到的新任务的指针值，而 `ecx` 中则保存着当前任务（本任务）的指针值。接着执行间接长

跳转到 `__tmp` 的指令 `ljmp`。长跳转到新任务 TSS 选择符的指令将忽略 `__tmp` 中未定义值的部分，CPU 将自动跳转到 TSS 段指定新任务中去执行，而本任务也就到此暂停执行。这也是我们无需设置结构变量 `__tmp` 中其他未定义部分的原因。参见第 5 章中图 2-22：任务切换操作示意图。

当一段时间之后，某个任务的 `ljmp` 指令又会跳转到本任务 TSS 段选择符，从而造成 CPU 切换回本任务，并从 `ljmp` 的下一条指令开始执行。此时 `ecx` 中含有本任务即当前任务的指针，因此我们可以使用该指针来检查它是否是最后（最近）一个使用过数学协处理器的任务。若本任务没有使用过协处理器则立刻退出，否则执行 `clts` 指令以复位控制寄存器 CR0 中的任务已切换标志 TS。每当任务切换时 CPU 都会设置该标志位，并且在执行协处理器指令之前测试该标志位。Linux 系统中的这种处理 TS 标志的方法可以让内核避免对协处理状态不必要的保存、恢复操作过程，从而提高了协处理器的执行性能。

14.28.2 代码注释

程序 14-25 linux/include/linux/sched.h

```

1  #ifndef SCHED\_H
2  #define SCHED\_H
3
4  #define NR\_TASKS 64 // 系统中同时最多任务（进程）数。
5  #define HZ 100 // 定义系统时钟滴答频率(1 百赫兹，每个滴答 10ms)
6
7  #define FIRST\_TASK task[0] // 任务 0 比较特殊，所以特意给它单独定义一个符号。
8  #define LAST\_TASK task[NR\_TASKS-1] // 任务数组中的最后一项任务。
9
10 #include <linux/head.h> // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
11 #include <linux/fs.h> // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
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 <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。
14
15 #if (NR\_OPEN > 32)
16 #error "Currently the close-on-exec-flags are in one word, max 32 files/proc"
17 #endif
18
19 // 这里定义了进程运行可能处的状态。
20 #define TASK\_RUNNING 0 // 进程正在运行或已准备就绪。
21 #define TASK\_INTERRUPTIBLE 1 // 进程处于可中断等待状态。
22 #define TASK\_UNINTERRUPTIBLE 2 // 进程处于不可中断等待状态，主要用于 I/O 操作等待。
23 #define TASK\_ZOMBIE 3 // 进程处于僵死状态，已经停止运行，但父进程还没发信号。
24 #define TASK\_STOPPED 4 // 进程已停止。
25
26 #ifndef NULL
27 #define NULL ((void *) 0) // 定义 NULL 为空指针。
28 #endif
29
30 // 复制进程的页目录页表。Linus 认为这是内核中最复杂的函数之一。（mm/memory.c, 105）
31 extern int copy\_page\_tables(unsigned long from, unsigned long to, long size);
32 // 释放页表所指定的内存块及页表本身。（mm/memory.c, 150）
33 extern int free\_page\_tables(unsigned long from, unsigned long size);
34
35 // 调度程序的初始化函数。（kernel/sched.c, 385）
36 extern void sched\_init(void);
37 // 进程调度函数。（kernel/sched.c, 104）
38 extern void schedule(void);

```

```

// 异常(陷阱)中断处理初始化函数, 设置中断调用门并允许中断请求信号。( kernel/traps.c, 181 )
34 extern void trap_init(void);
// 显示内核出错信息, 然后进入死循环。( kernel/panic.c, 16 )。
35 extern void panic(const char * str);
// 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
36 extern int tty_write(unsigned minor, char * buf, int count);
37
38 typedef int (*fn_ptr)(); // 定义函数指针类型。
39
// 下面是数学协处理器使用的结构, 主要用于保存进程切换时 i387 的执行状态信息。
40 struct i387_struct {
41     long cwd; // 控制字(Control word)。
42     long swd; // 状态字(Status word)。
43     long twd; // 标记字(Tag word)。
44     long fip; // 协处理器代码指针。
45     long fcs; // 协处理器代码段寄存器。
46     long foo; // 内存操作数的偏移位置。
47     long fos; // 内存操作数的段值。
48     long st_space[20]; /* 8*10 bytes for each FP-reg = 80 bytes */
49 }; // 8 个 10 字节的协处理器累加器。
50
// 任务状态段数据结构 (参见列表后的 TSS 信息)。
51 struct tss_struct {
52     long back_link; /* 16 high bits zero */
53     long esp0;
54     long ss0; /* 16 high bits zero */
55     long esp1;
56     long ssl; /* 16 high bits zero */
57     long esp2;
58     long ss2; /* 16 high bits zero */
59     long cr3;
60     long eip;
61     long eflags;
62     long eax, ecx, edx, ebx;
63     long esp;
64     long ebp;
65     long esi;
66     long edi;
67     long es; /* 16 high bits zero */
68     long cs; /* 16 high bits zero */
69     long ss; /* 16 high bits zero */
70     long ds; /* 16 high bits zero */
71     long fs; /* 16 high bits zero */
72     long gs; /* 16 high bits zero */
73     long ldt; /* 16 high bits zero */
74     long trace_bitmap; /* bits: trace 0, bitmap 16-31 */
75     struct i387_struct i387;
76 };
77
// 这里是任务 (进程) 数据结构, 或称为进程描述符。
// =====
// long state 任务的运行状态 (-1 不可运行, 0 可运行(就绪), >0 已停止)。
// long counter 任务运行时间计数(递减)(滴答数), 运行时间片。

```

```

// long priority          运行优先数。任务开始运行时 counter = priority, 越大运行越长。
// long signal            信号。是位图, 每个比特位代表一种信号, 信号值=位偏移值+1。
// struct sigaction sigaction[32] 信号执行属性结构, 对应信号将要执行的操作和标志信息。
// long blocked            进程信号屏蔽码 (对应信号位图)。
// -----
// int exit_code           任务执行停止的退出码, 其父进程会取。
// unsigned long start_code 代码段地址。
// unsigned long end_code   代码长度 (字节数)。
// unsigned long end_data   代码长度 + 数据长度 (字节数)。
// unsigned long brk        总长度 (字节数)。
// unsigned long start_stack 堆栈段地址。
// long pid                进程标识号 (进程号)。
// long father             父进程号。
// long pgrp               进程组号。
// long session            会话号。
// long leader             会话首领。
// unsigned short uid       用户标识号 (用户 id)。
// unsigned short euid      有效用户 id。
// unsigned short suid      保存的用户 id。
// unsigned short gid       组标识号 (组 id)。
// unsigned short egid      有效组 id。
// unsigned short sgid      保存的组 id。
// long alarm              报警定时值 (滴答数)。
// long utime              用户态运行时间 (滴答数)。
// long stime              系统态运行时间 (滴答数)。
// long cutime              子进程用户态运行时间。
// long cstime              子进程系统态运行时间。
// long start_time          进程开始运行时刻。
// unsigned short used_math 标志: 是否使用了协处理器。
// -----
// int tty                 进程使用 tty 的子设备号。-1 表示没有使用。
// unsigned short umask     文件创建属性屏蔽位。
// struct m_inode * pwd     当前工作目录 i 节点结构。
// struct m_inode * root    根目录 i 节点结构。
// struct m_inode * executable 执行文件 i 节点结构。
// unsigned long close_on_exec 执行时关闭文件句柄位标志。 (参见 include/fcntl.h)
// struct file * filp[NR_OPEN] 进程使用的文件表结构。
// -----
// struct desc_struct ldt[3] 本任务的局部表描述符。0-空, 1-代码段 cs, 2-数据和堆栈段 ds&ss。
// -----
// struct tss_struct tss    本进程的任务状态段信息结构。
// =====
78 struct task_struct {
79 /* these are hardcoded - don't touch */
80     long state;          /* -1 unrunnable, 0 runnable, >0 stopped */
81     long counter;
82     long priority;
83     long signal;
84     struct sigaction sigaction[32];
85     long blocked;        /* bitmap of masked signals */
86 /* various fields */
87     int exit_code;
88     unsigned long start_code, end_code, end_data, brk, start_stack;

```

```

89     long pid, father, pgrp, session, leader;
90     unsigned short uid, euid, suid;
91     unsigned short gid, egid, sgid;
92     long alarm;
93     long utime, stime, cutime, cstime, start_time;
94     unsigned short used_math;
95     /* file system info */
96     int tty;                /* -1 if no tty, so it must be signed */
97     unsigned short umask;
98     struct m_inode * pwd;
99     struct m_inode * root;
100    struct m_inode * executable;
101    unsigned long close_on_exec;
102    struct file * filp[NR_OPEN];
103    /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
104    struct desc_struct ldt[3];
105    /* tss for this task */
106    struct tss_struct tss;
107 };
108
109 /*
110  * INIT_TASK is used to set up the first task table, touch at
111  * your own risk!. Base=0, limit=0x9ffff (=640kB)
112  */
113 /*
114  * INIT_TASK 用于设置第 1 个任务表, 若想修改, 责任自负☺!
115  * 基址 Base = 0, 段长 limit = 0x9ffff (=640kB)。
116  */
117 // 对应上面任务结构的第 1 个任务的信息。
118 #define INIT_TASK \
119 /* state etc */ { 0, 15, 15, \      // state, counter, priority
120 /* signals */ 0, {}, 0, \      // signal, sigaction[32], blocked
121 /* ec, brk... */ 0, 0, 0, 0, 0, \  // exit_code, start_code, end_code, end_data, brk, start_stack
122 /* pid etc.. */ 0, -1, 0, 0, 0, \  // pid, father, pgrp, session, leader
123 /* uid etc */ 0, 0, 0, 0, 0, 0, \  // uid, euid, suid, gid, egid, sgid
124 /* alarm */ 0, 0, 0, 0, 0, 0, \  // alarm, utime, stime, cutime, cstime, start_time
125 /* math */ 0, \      // used_math
126 /* fs info */ -1, 0022, NULL, NULL, NULL, 0, \ // tty, umask, pwd, root, executable, close_on_exec
127 /* filp */ {NULL, }, \      // filp[20]
128 { \      // ldt[3]
129 {0, 0}, \
130 /* ldt */ {0x9f, 0xc0fa00}, \  // 代码长 640K, 基址 0x0, G=1, D=1, DPL=3, P=1 TYPE=0x0a
131 {0x9f, 0xc0f200}, \  // 数据长 640K, 基址 0x0, G=1, D=1, DPL=3, P=1 TYPE=0x02
132 }, \
133 /*tss*/ {0, PAGE_SIZE+(long)&init_task, 0x10, 0, 0, 0, 0, (long)&pg_dir, \  // tss
134 0, 0, 0, 0, 0, 0, 0, 0, \
135 0, 0, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, \
136 LDT(0), 0x80000000, \
137 {} \
138 }, \
139 }
140
141 extern struct task_struct *task[NR_TASKS];    // 任务指针数组。

```

```

137 extern struct task_struct *last_task_used_math; // 上一个使用过协处理器的进程。
138 extern struct task_struct *current;           // 当前进程结构指针变量。
139 extern long volatile jiffies;                 // 从开机开始算起的滴答数 (10ms/滴答)。
140 extern long startup_time;                     // 开机时间。从 1970:0:0:0 开始计时的秒数。
141
142 #define CURRENT_TIME (startup_time+jiffies/HZ) // 当前时间 (秒数)。
143
144 // 添加定时器函数 (定时时间 jiffies 滴答数, 定时到时调用函数*fn())。( kernel/sched.c, 272)
144 extern void add_timer(long jiffies, void (*fn)(void));
145 // 不可中断的等待睡眠。( kernel/sched.c, 151 )
145 extern void sleep_on(struct task_struct ** p);
146 // 可中断的等待睡眠。( kernel/sched.c, 167 )
146 extern void interruptible_sleep_on(struct task_struct ** p);
147 // 明确唤醒睡眠的进程。( kernel/sched.c, 188 )
147 extern void wake_up(struct task_struct ** p);
148
149 /*
150  * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
151  * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
152  */
153 /*
154  * 在 GDT 表中寻找第 1 个 TSS 的入口。0-没有用 nul, 1-代码段 cs, 2-数据段 ds, 3-系统调用 syscall
155  * 4-任务状态段 TSS0, 5-局部表 LDT0, 6-任务状态段 TSS1, 等。
156  */
157 // 从该英文注释可以猜想到, Linus 当时曾想把系统调用的代码专门放在 GDT 表中第 4 个独立的段中。
158 // 但后来并没有那样做, 于是就一直把 GDT 表中第 4 个描述符项 (上面 syscall 项) 闲置在一旁。
159 // 下面定义宏: 全局表中第 1 个任务状态段(TSS)描述符的选择符索引号。
153 #define FIRST_TSS_ENTRY 4
154 // 全局表中第 1 个局部描述符表(LDT)描述符的选择符索引号。
154 #define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
155 // 宏定义, 计算在全局表中第 n 个任务的 TSS 段描述符的选择符值 (偏移量)。
156 // 因每个描述符占 8 字节, 因此 FIRST_TSS_ENTRY<<3 表示该描述符在 GDT 表中的起始偏移位置。
157 // 因为每个任务使用 1 个 TSS 和 1 个 LDT 描述符, 共占用 16 字节, 因此需要 n<<4 来表示对应
158 // TSS 起始位置。该宏得到的值正好也是该 TSS 的选择符值。
155 #define TSS(n) (((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
156 // 宏定义, 计算在全局表中第 n 个任务的 LDT 段描述符的选择符值 (偏移量)。
156 #define LDT(n) (((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
157 // 宏定义, 把第 n 个任务的 TSS 段选择符加载到任务寄存器 TR 中。
157 #define ltr(n) __asm__("ltr %%ax"::"a" (TSS(n)))
158 // 宏定义, 把第 n 个任务的 LDT 段选择符加载到局部描述符表寄存器 LDTR 中。
158 #define lldt(n) __asm__("lldt %%ax"::"a" (LDT(n)))
159 // 取当前运行任务的任务号 (是任务数组中的索引值, 与进程号 pid 不同)。
160 // 返回: n - 当前任务号。用于 ( kernel/traps.c, 79)。
159 #define str(n) \
160 __asm__("str %%ax|n|t" \ // 将任务寄存器中 TSS 段的选择符复制到 ax 中。
161 "subl %2, %%eax|n|t" \ // (eax - FIRST_TSS_ENTRY*8) → eax
162 "shrl $4, %%eax" \ // (eax/16) → eax = 当前任务号。
163 : "=a" (n) \
164 : "a" (0), "i" (FIRST_TSS_ENTRY<<3))
165 /*
166  * switch_to(n) should switch tasks to task nr n, first
167  * checking that n isn't the current task, in which case it does nothing.
168  * This also clears the TS-flag if the task we switched to has used

```

```

169 * tha math co-processor latest.
170 */
/*
* switch_to(n)将切换当前任务到任务 nr, 即 n。首先检测任务 n 不是当前任务,
* 如果是则什么也不做退出。如果我们切换到的任务最近(上次运行)使用过数学
* 协处理器的话, 则还需复位控制寄存器 cr0 中的 TS 标志。
*/
// 跳转到一个任务的 TSS 段选择符组成的地址处会造成 CPU 进行任务切换操作。
// 输入: %0 - 指向__tmp; %1 - 指向__tmp.b 处, 用于存放新 TSS 的选择符;
// dx - 新任务 n 的 TSS 段选择符; ecx - 新任务 n 的任务结构指针 task[n]。
// 其中临时数据结构__tmp 用于组建 177 行远跳转 (far jump) 指令的操作数。该操作数由 4 字节偏移
// 地址和 2 字节的段选择符组成。因此__tmp 中 a 的值是 32 位偏移值, 而 b 的低 2 字节是新 TSS 段的
// 选择符(高 2 字节不用)。跳转到 TSS 段选择符会造成任务切换到该 TSS 对应的进程。对于造成任务
// 切换的长跳转, a 值无用。177 行上的内存间接跳转指令使用 6 字节操作数作为跳转目的地的长指针,
// 其格式为: jmp 16 位段选择符: 32 位偏移值。但在内存中操作数的表示顺序与这里正好相反。
// 任务切换回来之后, 在判断原任务上次执行是否使用过协处理器时, 是通过将原任务指针与保存在
// last_task_used_math 变量中的上次使用过协处理器任务指针进行比较而作出的, 参见文件
// kernel/sched.c 中有关 math_state_restore() 函数的说明。
171 #define switch_to(n) {\
172 struct {long a,b;} __tmp; \
173 __asm__( "cpl %%ecx, _current|n|t" \ // 任务 n 是当前任务吗?(current ==task[n]?)
174 "je 1f|n|t" \ // 是, 则什么都不做, 退出。
175 "movw %%dx, %1|n|t" \ // 将新任务 TSS 的 16 位选择符存入__tmp.b 中。
176 "xchgl %%ecx, _current|n|t" \ // current = task[n]; ecx = 被切换出的任务。
177 "ljmp %0|n|t" \ // 执行长跳转至*&__tmp, 造成任务切换。
// 在任务切换回来后会继续执行下面的语句。
178 "cpl %%ecx, _last_task_used_math|n|t" \ // 原任务上次使用过协处理器吗?
179 "jne 1f|n|t" \ // 没有则跳转, 退出。
180 "clts|n" \ // 原任务上次使用过协处理器, 则清 cr0 中的任务切换
181 "l:" \ // 标志 TS。
182 ":: "m" (*&__tmp.a), "m" (*&__tmp.b), \
183 "d" ( _TSS(n)), "c" ((long) task[n])); \
184 }
185 // 页面地址对准。(在内核代码中没有任何地方引用!!)
186 #define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)
187 // 设置位于地址 addr 处描述符中的各基地址字段(基地址是 base)。
// %0 - 地址 addr 偏移 2; %1 - 地址 addr 偏移 4; %2 - 地址 addr 偏移 7; edx - 基地址 base。
188 #define _set_base(addr, base) \
189 __asm__( "movw %%dx, %0|n|t" \ // 基址 base 低 16 位(位 15-0) → [addr+2]。
190 "rorl $16, %%edx|n|t" \ // edx 中基址高 16 位(位 31-16) → dx。
191 "movb %%dl, %1|n|t" \ // 基址高 16 位中的低 8 位(位 23-16) → [addr+4]。
192 "movb %%dh, %2" \ // 基址高 16 位中的高 8 位(位 31-24) → [addr+7]。
193 ":: "m" (*(addr)+2), \
194 "m" (*(addr)+4), \
195 "m" (*(addr)+7), \
196 "d" (base) \
197 : "dx") // 告诉 gcc 编译器 edx 寄存器中的值已被嵌入汇编程序改变了。
198 // 设置位于地址 addr 处描述符中的段限长字段(段长是 limit)。
// %0 - 地址 addr; %1 - 地址 addr 偏移 6 处; edx - 段长值 limit。
199 #define _set_limit(addr, limit) \

```



```

200 __asm__ ( "movw %%dx, %0\n\t" \           // 段长 limit 低 16 位(位 15-0) → [addr]。
201         "rorl $16, %%edx\n\t" \           // edx 中的段长高 4 位(位 19-16) → dl。
202         "movb %1, %%dh\n\t" \             // 取原[addr+6]字节 → dh, 其中高 4 位是些标志。
203         "andb $0xf0, %%dh\n\t" \          // 清 dh 的低 4 位(将存放段长的位 19-16)。
204         "orb %%dh, %%dl\n\t" \            // 将原高 4 位标志和段长的高 4 位(位 19-16)合成 1 字节,
205         "movb %%dl, %1" \                 // 并放会[addr+6]处。
206         :: "m" (*(addr)), \
207         "m" (*(addr+6)), \
208         "d" (limit) \
209         : "dx")
210
// 设置局部描述符表中 ldt 描述符的基地址字段。
211 #define set_base(ldt, base) set_base( ((char *)&(ldt)) , base )
// 设置局部描述符表中 ldt 描述符的段长字段。
212 #define set_limit(ldt, limit) set_limit( ((char *)&(ldt)) , (limit-1)>>12 )
213
// 从地址 addr 处描述符中取段基地址。功能与_set_base()正好相反。
// edx - 存放基地址(__base); %1 - 地址 addr 偏移 2; %2 - 地址 addr 偏移 4; %3 - addr 偏移 7。
214 #define get_base(addr) ({\
215 unsigned long __base; \
216 __asm__ ( "movb %3, %%dh\n\t" \           // 取[addr+7]处基址高 16 位的高 8 位(位 31-24) → dh。
217         "movb %2, %%dl\n\t" \           // 取[addr+4]处基址高 16 位的低 8 位(位 23-16) → dl。
218         "shll $16, %%edx\n\t" \          // 基址高 16 位移到 edx 中高 16 位处。
219         "movw %1, %%dx" \               // 取[addr+2]处基址低 16 位(位 15-0) → dx。
220         : "=d" (__base) \               // 从而 edx 中含有 32 位的段基地址。
221         : "m" (*(addr+2)), \
222         "m" (*(addr+4)), \
223         "m" (*(addr+7))) ; \
224 __base; })
225
// 取局部描述符表中 ldt 所指段描述符中的基地址。
226 #define get_base(ldt) get_base( ((char *)&(ldt)) )
227
// 取段选择符 segment 指定的描述符中的段限长值。
// 指令 lsl 是 Load Segment Limit 缩写。它从指定段描述符中取出分散的限长比特位拼成完整的
// 段限长值放入指定寄存器中。所得的段限长是实际字节数减 1, 因此这里还需要加 1 后才返回。
// %0 - 存放段长值(字节数); %1 - 段选择符 segment。
228 #define get_limit(segment) ({ \
229 unsigned long __limit; \
230 __asm__ ( "lsl %1, %0\n\tincl %0": "=r" (__limit): "r" (segment)); \
231 __limit; })
232
233 #endif
234

```

14.29 sys.h 文件

14.29.1 功能描述

sys.h 头文件列出了内核中所有系统调用函数的原型, 以及系统调用函数指针表。

14.29.2 代码注释

程序 14-26 linux/include/linux/sys.h

```

1 extern int sys_setup();           // 系统启动初始化设置函数。 (kernel/blk_drv/hd.c, 71)
2 extern int sys_exit();            // 程序退出。 (kernel/exit.c, 137)
3 extern int sys_fork();            // 创建进程。 (kernel/system_call.s, 208)
4 extern int sys_read();            // 读文件。 (fs/read_write.c, 55)
5 extern int sys_write();           // 写文件。 (fs/read_write.c, 83)
6 extern int sys_open();            // 打开文件。 (fs/open.c, 138)
7 extern int sys_close();           // 关闭文件。 (fs/open.c, 192)
8 extern int sys_waitpid();         // 等待进程终止。 (kernel/exit.c, 142)
9 extern int sys_creat();           // 创建文件。 (fs/open.c, 187)
10 extern int sys_link();            // 创建一个文件的硬连接。 (fs/namei.c, 721)
11 extern int sys_unlink();          // 删除一个文件名(或删除文件)。 (fs/namei.c, 663)
12 extern int sys_execve();          // 执行程序。 (kernel/system_call.s, 200)
13 extern int sys_chdir();           // 更改当前目录。 (fs/open.c, 75)
14 extern int sys_time();            // 取当前时间。 (kernel/sys.c, 102)
15 extern int sys_mknod();           // 建立块/字符特殊文件。 (fs/namei.c, 412)
16 extern int sys_chmod();           // 修改文件属性。 (fs/open.c, 105)
17 extern int sys_chown();           // 修改文件宿主和所属组。 (fs/open.c, 121)
18 extern int sys_break();           // (-kernel/sys.c, 21)
19 extern int sys_stat();            // 使用路径名取文件的状态信息。 (fs/stat.c, 36)
20 extern int sys_lseek();           // 重新定位读/写文件偏移。 (fs/read_write.c, 25)
21 extern int sys_getpid();          // 取进程 id。 (kernel/sched.c, 348)
22 extern int sys_mount();           // 安装文件系统。 (fs/super.c, 200)
23 extern int sys_umount();          // 卸载文件系统。 (fs/super.c, 167)
24 extern int sys_setuid();          // 设置进程用户 id。 (kernel/sys.c, 143)
25 extern int sys_getuid();          // 取进程用户 id。 (kernel/sched.c, 358)
26 extern int sys_stime();           // 设置系统时间日期。 (-kernel/sys.c, 148)
27 extern int sys_ptrace();          // 程序调试。 (-kernel/sys.c, 26)
28 extern int sys_alarm();           // 设置报警。 (kernel/sched.c, 338)
29 extern int sys_fstat();           // 使用文件句柄取文件的状态信息。 (fs/stat.c, 47)
30 extern int sys_pause();           // 暂停进程运行。 (kernel/sched.c, 144)
31 extern int sys_utime();           // 改变文件的访问和修改时间。 (fs/open.c, 24)
32 extern int sys_stty();            // 修改终端行设置。 (-kernel/sys.c, 31)
33 extern int sys_gtty();            // 取终端行设置信息。 (-kernel/sys.c, 36)
34 extern int sys_access();          // 检查用户对一个文件的访问权限。 (fs/open.c, 47)
35 extern int sys_nice();            // 设置进程执行优先权。 (kernel/sched.c, 378)
36 extern int sys_ftime();           // 取日期和时间。 (-kernel/sys.c, 16)
37 extern int sys_sync();            // 同步高速缓冲与设备中数据。 (fs/buffer.c, 44)
38 extern int sys_kill();            // 终止一个进程。 (kernel/exit.c, 60)
39 extern int sys_rename();          // 更改文件名。 (-kernel/sys.c, 41)
40 extern int sys_mkdir();           // 创建目录。 (fs/namei.c, 463)
41 extern int sys_rmdir();           // 删除目录。 (fs/namei.c, 587)
42 extern int sys_dup();             // 复制文件句柄。 (fs/fcntl.c, 42)
43 extern int sys_pipe();            // 创建管道。 (fs/pipe.c, 71)
44 extern int sys_times();           // 取运行时间。 (kernel/sys.c, 156)
45 extern int sys_prof();            // 程序执行时间区域。 (-kernel/sys.c, 46)
46 extern int sys_brk();             // 修改数据段长度。 (kernel/sys.c, 168)
47 extern int sys_setgid();          // 设置进程组 id。 (kernel/sys.c, 72)
48 extern int sys_getgid();          // 取进程组 id。 (kernel/sched.c, 368)
49 extern int sys_signal();          // 信号处理。 (kernel/signal.c, 48)

```

```

50 extern int sys\_geteuid(); // 取进程有效用户 id。 (kenrl/sched.c, 363)
51 extern int sys\_getegid(); // 取进程有效组 id。 (kenrl/sched.c, 373)
52 extern int sys\_acct(); // 进程记帐。 (-kernel/sys.c, 77)
53 extern int sys\_phys(); // (-kernel/sys.c, 82)
54 extern int sys\_lock(); // (-kernel/sys.c, 87)
55 extern int sys\_ioctl(); // 设备控制。 (fs/ioctl.c, 30)
56 extern int sys\_fcntl(); // 文件句柄操作。 (fs/fcntl.c, 47)
57 extern int sys\_mpx(); // (-kernel/sys.c, 92)
58 extern int sys\_setpgid(); // 设置进程组 id。 (kernel/sys.c, 181)
59 extern int sys\_ulimit(); // (-kernel/sys.c, 97)
60 extern int sys\_uname(); // 显示系统信息。 (kernel/sys.c, 216)
61 extern int sys\_umask(); // 取默认文件创建属性码。 (kernel/sys.c, 230)
62 extern int sys\_chroot(); // 改变根目录。 (fs/open.c, 90)
63 extern int sys\_ustat(); // 取文件系统信息。 (fs/open.c, 19)
64 extern int sys\_dup2(); // 复制文件句柄。 (fs/fcntl.c, 36)
65 extern int sys\_getppid(); // 取父进程 id。 (kernel/sched.c, 353)
66 extern int sys\_getpgrp(); // 取进程组 id, 等于 getpgid(0)。 (kernel/sys.c, 201)
67 extern int sys\_setsid(); // 在新会话中运行程序。 (kernel/sys.c, 206)
68 extern int sys\_sigaction(); // 改变信号处理过程。 (kernel/signal.c, 63)
69 extern int sys\_sgetmask(); // 取信号屏蔽码。 (kernel/signal.c, 15)
70 extern int sys\_ssetmask(); // 设置信号屏蔽码。 (kernel/signal.c, 20)
71 extern int sys\_setreuid(); // 设置真实与/或有效用户 id。 (kernel/sys.c, 118)
72 extern int sys\_setregid(); // 设置真实与/或有效组 id。 (kernel/sys.c, 51)
73 // 系统调用函数指针表。用于系统调用中断处理程序(int 0x80), 作为跳转表。
74 fn\_ptr sys\_call\_table[] = { sys\_setup, sys\_exit, sys\_fork, sys\_read,
75 sys\_write, sys\_open, sys\_close, sys\_waitpid, sys\_creat, sys\_link,
76 sys\_unlink, sys\_execve, sys\_chdir, sys\_time, sys\_mknod, sys\_chmod,
77 sys\_chown, sys\_break, sys\_stat, sys\_lseek, sys\_getpid, sys\_mount,
78 sys\_umount, sys\_setuid, sys\_getuid, sys\_stime, sys\_ptrace, sys\_alarm,
79 sys\_fstat, sys\_pause, sys\_ftime, sys\_sstk, sys\_gtty, sys\_access,
80 sys\_nice, sys\_ftime, sys\_sync, sys\_kill, sys\_rename, sys\_mkdir,
81 sys\_rmdir, sys\_dup, sys\_pipe, sys\_times, sys\_prof, sys\_brk, sys\_setgid,
82 sys\_getgid, sys\_signal, sys\_geteuid, sys\_getegid, sys\_acct, sys\_phys,
83 sys\_lock, sys\_ioctl, sys\_fcntl, sys\_mpx, sys\_setpgid, sys\_ulimit,
84 sys\_uname, sys\_umask, sys\_chroot, sys\_ustat, sys\_dup2, sys\_getppid,
85 sys\_getpgrp, sys\_setsid, sys\_sigaction, sys\_sgetmask, sys\_ssetmask,
86 sys\_setreuid, sys\_setregid };
87

```

14.30 tty.h 文件

14.30.1 功能描述

终端数据结构和常量定义。

14.30.2 代码注释

程序 14-27 linux/include/linux/tty.h

```

1 /*
2  * 'tty.h' defines some structures used by tty_io.c and some defines.

```

```

3  *
4  * NOTE! Don't touch this without checking that nothing in rs_io.s or
5  * con_io.s breaks. Some constants are hardwired into the system (mainly
6  * offsets into 'tty_queue'
7  */
8
9  /*
10 * 'tty.h' 中定义了 tty_io.c 程序使用的某些结构和其他一些定义。
11 *
12 * 注意！在修改这里的定义时，一定要检查 rs_io.s 或 con_io.s 程序中不会出现问题。
13 * 在系统中有些常量是直接写在程序中的（主要是一些 tty_queue 中的偏移值）。
14 */
15 #ifndef TTY_H
16 #define TTY_H
17
18 #include <termios.h>          // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
19
20 #define TTY_BUF_SIZE 1024    // tty 缓冲区（缓冲队列）大小。
21
22 // tty 字符缓冲队列数据结构。用于 tty_struct 结构中的读、写和辅助（规范）缓冲队列。
23 struct tty_queue {
24     unsigned long data;      // 队列缓冲区中含有字符行数（不是当前字符数）。
25                               // 对于串口终端，则存放串行端口地址。
26     unsigned long head;      // 缓冲区中数据头指针。
27     unsigned long tail;      // 缓冲区中数据尾指针。
28     struct task_struct * proc_list; // 等待进程列表。
29     char buf[TTY_BUF_SIZE];    // 队列的缓冲区。
30 };
31
32 // 以下定义了 tty 等待队列中缓冲区操作宏函数。（tail 在前，head 在后，参见 tty_io.c 的图）。
33 // a 缓冲区指针前移 1 字节，若已超出缓冲区右侧，则指针循环。
34 #define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
35 // a 缓冲区指针后退 1 字节，并循环。
36 #define DEC(a) ((a) = ((a)-1) & (TTY_BUF_SIZE-1))
37 // 清空指定队列的缓冲区。
38 #define EMPTY(a) ((a).head == (a).tail)
39 // 缓冲区还可存放字符的长度（空闲区长度）。
40 #define LEFT(a) (((a).tail-(a).head-1)&(TTY_BUF_SIZE-1))
41 // 缓冲区中最后一个位置。
42 #define LAST(a) ((a).buf[(TTY_BUF_SIZE-1)&((a).head-1)])
43 // 缓冲区满（如果为 1 的话）。
44 #define FULL(a) (!LEFT(a))
45 // 缓冲区中已存放字符的长度。
46 #define CHARS(a) (((a).head-(a).tail)&(TTY_BUF_SIZE-1))
47 // 从 queue 队列项缓冲区中取一字符（从 tail 处，并且 tail+=1）。
48 #define GETCH(queue, c) \
49 (void)({c=(queue).buf[(queue).tail];INC((queue).tail);})
50 // 往 queue 队列项缓冲区中放置一字符（在 head 处，并且 head+=1）。
51 #define PUTCH(c, queue) \
52 (void)({(queue).buf[(queue).head]=c;INC((queue).head);})
53
54 // 判断终端键盘字符类型。
55 #define INTR_CHAR(tty) ((tty)->termios.c_cc[VINTR]) // 中断符。发中断信号 SIGINT。

```






```

37 #define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT]) // 退出符。发退出信号 SIGQUIT。
38 #define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE]) // 删除符。擦除一个字符。
39 #define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL]) // 删除行。删除一行字符。
40 #define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF]) // 文件结束符。
41 #define START_CHAR(tty) ((tty)->termios.c_cc[VSTART]) // 开始符。恢复输出。
42 #define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP]) // 停止符。停止输出。
43 #define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP]) // 挂起符。发挂起信号 SIGTSTP。
44
// tty 数据结构。
45 struct tty_struct {
46     struct termios termios; // 终端 io 属性和控制字符数据结构。
47     int pgrp; // 所属进程组。
48     int stopped; // 停止标志。
49     void (*write)(struct tty_struct * tty); // tty 写函数指针。
50     struct tty_queue read_q; // tty 读队列。
51     struct tty_queue write_q; // tty 写队列。
52     struct tty_queue secondary; // tty 辅助队列(存放规范模式字符序列),
53     }; // 可称为规范(熟)模式队列。
54
55 extern struct tty_struct tty_table[]; // tty 结构数组。
56
// 这里给出了终端 termios 结构中可更改的特殊字符数组 c_cc[] 的初始值。该 termios 结构
// 定义在 include/termios.h 中。如果定义了 _POSIX_VDISABLE (\0), 那么当某一项值等于
// _POSIX_VDISABLE 的值时, 表示禁止使用相应的特殊字符。[8 进制值]
57 /*      intr=^C      quit=^_      erase=del      kill=^U
58      eof=^D      vtime=\0      vmin=\1      sxtc=\0
59      start=^Q      stop=^S      susp=^Z      eol=\0
60      reprint=^R      discard=^U      werase=^W      lnext=^V
61      eol2=\0
62 */
/* 中断 intr=^C      退出 quit=^_      删除 erase=del      终止 kill=^U
* 文件结束 eof=^D      vtime=\0      vmin=\1      sxtc=\0
* 开始 start=^Q      停止 stop=^S      挂起 susp=^Z      行结束 eol=\0
* 重显 reprint=^R      丢弃 discard=^U      werase=^W      lnext=^V
* 行结束 eol2=\0
*/
63 #define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"
64
65 void rs_init(void); // 异步串行通信初始化。(kernel/chr_drv/serial.c, 37)
66 void con_init(void); // 控制终端初始化。(kernel/chr_drv/console.c, 617)
67 void tty_init(void); // tty 初始化。(kernel/chr_drv/tty_io.c, 105)
68
69 int tty_read(unsigned c, char * buf, int n); // (kernel/chr_drv/tty_io.c, 230)
70 int tty_write(unsigned c, char * buf, int n); // (kernel/chr_drv/tty_io.c, 290)
71
72 void rs_write(struct tty_struct * tty); // (kernel/chr_drv/serial.c, 53)
73 void con_write(struct tty_struct * tty); // (kernel/chr_drv/console.c, 445)
74
75 void copy_to_cooked(struct tty_struct * tty); // (kernel/chr_drv/tty_io.c, 145)
76
77 #endif
78

```

14.31 include/sys/目录中的文件

列表 14-4 linux/include/sys/目录下的文件

名称	大小	最后修改时间 (GMT)	说明
 stat.h	1304 bytes	1991-09-17 15:02:48	m
 times.h	200 bytes	1991-09-17 15:03:06	m
 types.h	805 bytes	1991-09-17 15:02:55	m
 utsname.h	234 bytes	1991-09-17 15:03:23	m
 wait.h	560 bytes	1991-09-17 15:06:07	m

14.32 stat.h 文件

14.32.1 功能描述

该头文件说明了函数 stat()返回的数据及其结构类型，以及一些属性操作测试宏、函数原型。

14.32.2 代码注释

程序 14-28 linux/include/sys/stat.h

```

1 #ifndef SYS_STAT_H
2 #define SYS_STAT_H
3
4 #include <sys/types.h>
5
6 struct stat {
7     dev_t    st_dev;      // 含有文件的设备号。
8     ino_t    st_ino;      // 文件 i 节点号。
9     umode_t  st_mode;     // 文件类型和属性（见下面）。
10    nlink_t  st_nlink;    // 指定文件的连接数。
11    uid_t    st_uid;      // 文件的用户（标识）号。
12    gid_t    st_gid;      // 文件的组号。
13    dev_t    st_rdev;     // 设备号（如果文件是特殊的字符文件或块文件）。
14    off_t    st_size;     // 文件大小（字节数）（如果文件是常规文件）。
15    time_t   st_atime;    // 上次（最后）访问时间。
16    time_t   st_mtime;    // 最后修改时间。
17    time_t   st_ctime;    // 最后节点修改时间。
18 };
19
// 定义 st_mode 值的符号名称，这些值均用八进制表示。 参见第 9 章文件系统中图 9-5（i 节点
// 属性字段内容）。为便于记忆，这些符号名称均为一些英文单词的首字母或缩写组合而成。例
// 如名称 S_IFMT 的每个字母分别代表单词 State、Inode、File、Mask 和 Type；而名称 S_IFREG
// 则是 State、Inode、File 和 REGular 几个大写字母的组合；名称 S_IRWXU 是 State、Inode、
// Read、Write、eXecute 和 User 的组合。其它的名称可以此类推。

```

```

// 文件类型:
20 #define S_IFMT 00170000 // 文件类型屏蔽码 (8 进制表示)。
21 #define S_IFREG 0100000 // 常规文件。
22 #define S_IFBLK 0060000 // 块特殊 (设备) 文件, 如磁盘 dev/fd0。
23 #define S_IFDIR 0040000 // 目录文件。
24 #define S_IFCHR 0020000 // 字符设备文件。
25 #define S_IFIFO 0010000 // FIFO 特殊文件。
// 文件属性位:
// S_ISUID 用于测试文件的 set-user-ID 标志是否置位。若该标志置位, 则当执行该文件时, 进程的
// 有效用户 ID 将被设置为该文件宿主的用户 ID。S_ISGID 则是针对组 ID 进行相同处理。
26 #define S_ISUID 0004000 // 执行时设置用户 ID (set-user-ID)。
27 #define S_ISGID 0002000 // 执行时设置组 ID (set-group-ID)。
28 #define S_ISVTX 0001000 // 对于目录, 受限删除标志。
29
30 #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG) // 测试是否常规文件。
31 #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR) // 是否目录文件。
32 #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR) // 是否字符设备文件。
33 #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK) // 是否块设备文件。
34 #define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) // 是否 FIFO 特殊文件。
35
// 文件访问权限:
36 #define S_IRWXU 00700 // 宿主可以读、写、执行/搜索 (名称最后字母代表 User)。
37 #define S_IRUSR 00400 // 宿主读许可 (最后 3 个字母代表 User)。
38 #define S_IWUSR 00200 // 宿主写许可。
39 #define S_IXUSR 00100 // 宿主执行/搜索许可。
40
41 #define S_IRWXG 00070 // 组成员可以读、写、执行/搜索 (名称最后字母代表 Group)。
42 #define S_IRGRP 00040 // 组成员读许可 (最后 3 个字母代表 Group)。
43 #define S_IWGRP 00020 // 组成员写许可。
44 #define S_IXGRP 00010 // 组成员执行/搜索许可。
45
46 #define S_IRWXO 00007 // 其他人读、写、执行/搜索许可 (名称最后字母 O 代表 Other)。
47 #define S_IROTH 00004 // 其他人读许可 (最后 3 个字母代表 Other)。
48 #define S_IWOTH 00002 // 其他人写许可。
49 #define S_IXOTH 00001 // 其他人执行/搜索许可。
50
51 extern int chmod(const char *path, mode_t mode); // 修改文件属性。
52 extern int fstat(int fildes, struct stat *stat_buf); // 取指定文件句柄的文件状态信息。
53 extern int mkdir(const char *path, mode_t mode); // 创建目录。
54 extern int mkfifo(const char *path, mode_t mode); // 创建管道文件。
55 extern int stat(const char *filename, struct stat *stat_buf); // 取指定文件名的文件状态信息。
56 extern mode_t umask(mode_t mask); // 设置属性屏蔽码。
57
58 #endif
59

```

14.33 times.h 文件

14.33.1 功能描述

该头文件中主要定义了文件访问与修改时间结构 `tms`。它将由 `times()` 函数返回。其中 `time_t` 是在 `sys/types.h` 中定义的。还定义了一个函数原型 `times()`。

14.33.2 代码注释

程序 14-29 linux/include/sys/times.h

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

14.34 types.h 文件

14.34.1 功能描述

`types.h` 头文件中定义了基本的数据类型。所有的类型均定义为适当的数学类型长度。另外，`size_t` 是无符号整数类型，`off_t` 是扩展的符号整数类型，`pid_t` 是符号整数类型。

14.34.2 代码注释

程序 14-30 linux/include/sys/types.h

```
1 #ifndef SYS\_TYPES\_H
2 #define SYS\_TYPES\_H
3
4 #ifndef SIZE\_T
5 #define SIZE\_T
6 typedef unsigned int size\_t;    // 用于对象的大小（长度）。
7 #endif
8
```



```

9 #ifndef TIME T
10 #define TIME T
11 typedef long time t;           // 用于时间（以秒计）。
12 #endif
13
14 #ifndef PTRDIFF T
15 #define PTRDIFF T
16 typedef long ptrdiff t;
17 #endif
18
19 #ifndef NULL
20 #define NULL ((void *) 0)
21 #endif
22
23 typedef int pid t;           // 用于进程号和进程组号。
24 typedef unsigned short uid t; // 用于用户号（用户标识号）。
25 typedef unsigned char gid t; // 用于组号。
26 typedef unsigned short dev t; // 用于设备号。
27 typedef unsigned short ino t; // 用于文件序列号。
28 typedef unsigned short mode t; // 用于某些文件属性。
29 typedef unsigned short umode t; //
30 typedef unsigned char nlink t; // 用于连接计数。
31 typedef int daddr t;
32 typedef long off t;           // 用于文件长度（大小）。
33 typedef unsigned char u\_char; // 无符号字符类型。
34 typedef unsigned short ushort; // 无符号短整数类型。
35
36 typedef struct { int quot,rem; } div t; // 用于 DIV 操作。
37 typedef struct { long quot,rem; } ldiv t; // 用于长 DIV 操作。
38
39 // 文件系统参数结构，用于 ustat() 函数。最后两个字段未使用，总是返回 NULL 指针。
40 struct ustat {
41     daddr t f\_tfree;           // 系统总空闲块数。
42     ino t f\_tinode;           // 总空闲 i 节点数。
43     char f\_fname[6];           // 文件系统名称。
44     char f\_fpack[6];           // 文件系统压缩名称。
45 };
46 #endif
47

```

14.35 utsname.h 文件

14.35.1 功能描述

`utsname.h` 是系统名称结构头文件。其中定义了 `utsname` 结构以及函数原型 `uname()`。该函数利用 `utsname` 结构中的信息给出系统标识、版本号以及硬件类型等信息。POSIX 标准中要求字符数组长度应该是不指定的，但是其中存储的数据需以 `null` 终止。因此该版内核的 `utsname` 结构定义不符合要求（字符数组长度都被定义为 9）。另外，名称 `utsname` 是 Unix Timesharing System name 的缩写。

14.35.2 代码注释

程序 14-31 linux/include/sys/utsname.h

```

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

```

14.36 wait.h 文件

14.36.1 功能描述

该头文件描述了进程等待时信息。包括一些符号常数和 `wait()`、`waitpid()` 函数原型声明。

14.36.2 代码注释

程序 14-32 linux/include/sys/wait.h

```

1 #ifndef SYS\_WAIT\_H
2 #define SYS\_WAIT\_H
3
4 #include <sys/types.h>
5
6 #define LOW(v)      ((v) & 0377)    // 取低字节（8 进制表示）。
7 #define HIGH(v)    (((v) >> 8) & 0377) // 取高字节。
8
9 /* options for waitpid, WUNTRACED not supported */
10 /* waitpid 的选项，其中 WUNTRACED 未被支持 */
11 // [ 注：其实 0.11 内核已经支持 WUNTRACED 选项。上面这条注释应该是以前内核版本遗留下来的。 ]
12 // 以下常数符号是函数 waitpid(pid_t pid, long *stat_addr, int options) 中 options 使用的选项。
13 #define WNOHANG      1    // 如果没有状态也不要挂起，并立刻返回。
14 #define WUNTRACED    2    // 报告停止执行的子进程状态。
15
16 // 以下宏定义用于判断 waitpid() 函数返回的状态字（第 20、21 行的参数 *stat_loc）的含义。
17 #define WIFEXITED(s)  (!((s)&0xFF)) // 如果子进程正常退出，则为真。

```


```
14 #define WIFSTOPPED(s) (((s)&0xFF)==0x7F) // 如果子进程正停止着，则为 true。
15 #define WEXITSTATUS(s) (((s)>>8)&0xFF) // 返回退出状态。
16 #define WTERMSIG(s) ((s)&0x7F) // 返回导致进程终止的信号值（信号量）。
17 #define WSTOPSIG(s) (((s)>>8)&0xFF) // 返回导致进程停止的信号值。
18 #define WIFSIGNALED(s) (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) // 如果由于未捕捉信号而
// 导致子进程退出则为真。
19
// wait() 和 waitpid() 函数允许进程获取与其子进程之一的状态信息。各种选项允许获取已经终止或
// 停止的子进程状态信息。如果存在两个或两个以上子进程的状态信息，则报告的顺序是不指定的。
// wait() 将挂起当前进程，直到其子进程之一退出（终止），或者收到要求终止该进程的信号，
// 或者是需要调用一个信号句柄（信号处理程序）。
// waitpid() 挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，
// 或者是需要调用一个信号句柄（信号处理程序）。
// 如果 pid=-1, options=0, 则 waitpid() 的作用与 wait() 函数一样。否则其行为将随 pid 和 options
// 参数的不同而不同。（参见 kernel/exit.c, 142）
// 参数 pid 是进程号；*stat_loc 是保存状态信息位置的指针；options 是等待选项，见第 10, 11 行。
20 pid_t wait(int *stat_loc);
21 pid_t waitpid(pid_t pid, int *stat_loc, int options);
22
23 #endif
24
```

第15章 库文件(lib)

c 语言的函数库（library）文件是一些可重用程序模块集合，而 Linux 内核库文件则是编译时专门供内核使用的一些内核常用函数的组合。下面列表中的 c 文件就是构成内核库文件中模块的程序，主要包括退出函数 `_exit()`、关闭文件函数 `close()`、复制文件描述符函数 `dup()`、文件打开函数 `open()`、写文件函数 `write()`、执行程序函数 `execve()`、内存分配函数 `malloc()`、等待子进程状态函数 `wait()`、创建会话系统调用 `setsid()` 以及在 `include/string.h` 中实现的所有字符串操作函数。

除了一个由 Tytso 编制的 `malloc.c` 程序较长以外，其他程序都什么短小，有的只有一二行代码。基本都是直接调用系统中断调用实现其功能。

列表 15-1 /linux/lib/目录中的文件

文件名	文件长度	最后修改时间 (GMT)	说明
 Makefile	2602 bytes	1991-12-02 03:16:05	
 _exit.c	198 bytes	1991-10-02 14:16:29	
 close.c	131 bytes	1991-10-02 14:16:29	
 ctype.c	1202 bytes	1991-10-02 14:16:29	
 dup.c	127 bytes	1991-10-02 14:16:29	
 errno.c	73 bytes	1991-10-02 14:16:29	
 execve.c	170 bytes	1991-10-02 14:16:29	
 malloc.c	7469 bytes	1991-12-02 03:15:20	
 open.c	389 bytes	1991-10-02 14:16:29	
 setsid.c	128 bytes	1991-10-02 14:16:29	
 string.c	177 bytes	1991-10-02 14:16:29	
 wait.c	253 bytes	1991-10-02 14:16:29	
 write.c	160 bytes	1991-10-02 14:16:29	

在编译内核阶段，`Makefile` 中的相关指令会把以上这些程序编译成 `.o` 模块，然后组建成 `lib.a` 库文件形式并链接到内核模块中。与通常编译环境提供的各种库文件不同（例如 `libc.a`、`libufc.a` 等），这个库中的函数主要用于内核初始化阶段的 `init/main.c` 程序，为其在用户态执行的 `init()` 函数提供支持。因此所包含的函数很少，也特别简单。但它与一般库文件的实现方式完全相同。

创建函数库通常使用命令 `ar`（`archive` – 归档缩写）。例如要创建一个含有 3 个模块 `a.o`、`b.o` 和 `c.o` 的函数库 `libmine.a`，则需要执行如下命令：

```
ar -rc libmine.a a.o b.o c.o d.o
```

若要往这个库文件中添加函数模块 `dup.o`，则可执行以下命令

```
ar -rs dup.o
```

15.1 Makefile 文件

15.1.1 功能描述

组建内核函数库的 Makefile 文件。

15.1.2 代码注释

程序 15-1 linux/lib/Makefile

```

1 #
2 # Makefile for some libs needed in the kernel.
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 # 内核需要用到 libs 库文件程序的 Makefile。
9 #
10 # 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
11 # 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）。
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 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
19          -finline-functions -mstring-insns -nostdinc -I../include
20 # 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；
21 # -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
22 # 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
23 # 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；
24 # -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)。
25
26 CPP      =gcc -E -nostdinc -I../include
27 # 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
28 # 出设备或指定的输出文件中；-nostdinc -I../include 同前。
29
30 # 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
31 # 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
32 # 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
33 # 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，
34 # $< 代表第一个先决条件，这里即是符合条件 *.c 的文件。
35 # 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
36 # 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
37 # 是 c 文件，则可直接使用第 3 个规则。
38
39 .c.s:
40     $(CC) $(CFLAGS) \
41     -S -o $.s $<
42 # 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。
43
44 .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 # 下面定义目标文件变量 OBJS。
28 OBJS = ctype.o _exit.o open.o close.o errno.o write.o dup.o setsid.o \
29     execve.o wait.o string.o malloc.o
30
31 # 在有了先决条件 OBJS 后使用下面的命令连接成目标 lib.a 库文件。
32 # 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,
33 # 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
34 # 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
35 # 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
36 # 使用命令 "ar s" 等同于对一个归档文件执行命令 ranlib。
37 lib.a: $(OBJS)
38     $(AR) rcs lib.a $(OBJS)
39
40 sync
41
42 # 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
43 # 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。
44 clean:
45     rm -f core *.o *.a tmp_make
46     for i in *.c;do rm -f `basename $$i .c`.s;done
47
48 # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下:
49 # 使用字符串编辑程序 sed 对 Makefile 文件 (即是本文件) 进行处理, 输出为删除 Makefile
50 # 文件中 '### Dependencies' 行后面的所有行 (下面从 45 开始的行), 并生成 tmp_make
51 # 临时文件 (39 行的作用)。然后对 kernel/blk_drv/目录下的每个 C 文件执行 gcc 预处理操作。
52 # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则, 并且这些规则符合 make 语法。
53 # 对于每一个源文件, 预处理程序输出一个 make 规则, 其结果形式是相应源程序文件的目标
54 # 文件名加上其依赖关系—该源文件中包含的所有头文件列表。把预处理结果都添加到临时
55 # 文件 tmp_make 中, 然后将该临时文件复制成新的 Makefile 文件。
56 dep:
57     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
58     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,\'`"; \
59         $(CPP) -M $$i;done) >> tmp_make
60     cp tmp_make Makefile
61
62 ### Dependencies:
63 _exit.s _exit.o : _exit.c ../include/unistd.h ../include/sys/stat.h \
64     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
65     ../include/utime.h
66 close.s close.o : close.c ../include/unistd.h ../include/sys/stat.h \
67     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
68     ../include/utime.h
69 ctype.s ctype.o : ctype.c ../include/ctype.h
70 dup.s dup.o : dup.c ../include/unistd.h ../include/sys/stat.h \
71     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
72     ../include/utime.h
73 errno.s errno.o : errno.c
74 execve.s execve.o : execve.c ../include/unistd.h ../include/sys/stat.h \
75     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \

```

```

58  ../include/utime.h
59 malloc.s malloc.o : malloc.c ../include/linux/kernel.h ../include/linux/mm.h \
60  ../include/asm/system.h
61 open.s open.o : open.c ../include/unistd.h ../include/sys/stat.h \
62  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
63  ../include/utime.h ../include/stdarg.h
64 setsid.s setsid.o : setsid.c ../include/unistd.h ../include/sys/stat.h \
65  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
66  ../include/utime.h
67 string.s string.o : string.c ../include/string.h
68 wait.s wait.o : wait.c ../include/unistd.h ../include/sys/stat.h \
69  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
70  ../include/utime.h ../include/sys/wait.h
71 write.s write.o : write.c ../include/unistd.h ../include/sys/stat.h \
72  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
73  ../include/utime.h

```

15.2 _exit.c 程序

15.2.1 功能描述

程序调用内核的退出系统调用函数。

15.2.2 代码注释

程序 15-2 linux/lib/_exit.c

```

1  /*
2   *  linux/lib/_exit.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #define  __LIBRARY__          // 定义一个符号常量，见下行说明。
8  #include <unistd.h>          //  Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                   //  若定义了__LIBRARY__，则还含系统调用号和内嵌汇编 syscall0() 等。
9
10  // 内核使用的程序(退出)终止函数。
11  // 直接调用系统中断 int 0x80，功能号__NR_exit。
12  // 参数：exit_code - 退出码。
13  // 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 gcc 的函数属性说明：void do_exit(int error_code) __attribute__((noreturn));
16  volatile void __exit(int exit_code)
17  {
18      // %0 - eax(系统调用号__NR_exit); %1 - ebx(退出码 exit_code)。
19      __asm__ ("int $0x80::: \"a\" (__NR_exit), \"b\" (exit_code));
20  }

```

15.2.3 相关信息

参见 include/unistd.h 中的说明。

15.3 close.c 程序

15.3.1 功能描述

close.c 文件中定义了文件关闭函数 close()。

15.3.2 代码注释

程序 15-3 linux/lib/close.c

```

1  /*
2  *  linux/lib/close.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #define  __LIBRARY__
8  #include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编 syscall0()等。
9
10 // 关闭文件函数。
11 // 下面该调用宏函数对应：int close(int fd)。直接调用了系统中断 int 0x80，参数是__NR_close。
12 // 其中 fd 是文件描述符。
13 __syscall1(int, close, int, fd)
14

```

15.4 ctype.c 程序

15.4.1 功能描述

该程序用于为 ctype.h 提供辅助的数组结构数据，用于对字符进行类型判断。

15.4.2 代码注释

程序 15-4 linux/lib/ctype.c

```

1  /*
2  *  linux/lib/ctype.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
8
9  char  _ctmp;               // 一个临时字符变量，供 ctype.h 文件中转换字符宏函数使用。
10

```

// 字符特性数组(表), 定义了各个字符对应的属性, 这些属性类型(如_C等)在 ctype.h 中定义。

// 用于判断字符是控制字符(_C)、大写字母(_U)、小写字母(_L)等所属类型。

```

10 unsigned char _ctype[] = {0x00,          /* EOF */
11  _C, _C, _C, _C, _C, _C, _C, _C,        /* 0-7 */
12  _C, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, /* 8-15 */
13  _C, _C, _C, _C, _C, _C, _C, _C,        /* 16-23 */
14  _C, _C, _C, _C, _C, _C, _C, _C,        /* 24-31 */
15  _S|_SP, _P, _P, _P, _P, _P, _P, _P,    /* 32-39 */
16  _P, _P, _P, _P, _P, _P, _P, _P,        /* 40-47 */
17  _D, _D, _D, _D, _D, _D, _D, _D,        /* 48-55 */
18  _D, _D, _P, _P, _P, _P, _P, _P,        /* 56-63 */
19  _P, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, /* 64-71 */
20  _U, _U, _U, _U, _U, _U, _U, _U,        /* 72-79 */
21  _U, _U, _U, _U, _U, _U, _U, _U,        /* 80-87 */
22  _U, _U, _U, _P, _P, _P, _P, _P,        /* 88-95 */
23  _P, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, /* 96-103 */
24  _L, _L, _L, _L, _L, _L, _L, _L,        /* 104-111 */
25  _L, _L, _L, _L, _L, _L, _L, _L,        /* 112-119 */
26  _L, _L, _L, _P, _P, _P, _P, _C,        /* 120-127 */
27  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 128-143 */
28  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 144-159 */
29  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 160-175 */
30  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 176-191 */
31  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 192-207 */
32  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 208-223 */
33  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 224-239 */
34  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 240-255 */
35
36

```

15.5 dup.c 程序

15.5.1 功能描述

该程序包括一个创建文件描述符拷贝的函数 dup()。在成功返回之后, 新的和原来的描述符可以交替使用。它们共享锁定、文件读写指针以及文件标志。例如, 如果文件读写位置指针被其中一个描述符使用 lseek()修改过之后, 则对于另一个描述符来讲, 文件读写指针也被改变。该函数使用数值最小的未使用描述符来建立新描述符。但是这两个描述符并不共享执行时关闭标志(close-on-exec)。

15.5.2 代码注释

程序 15-5 linux/lib/dup.c

```

1  /*
2  *  linux/lib/dup.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6

```

```

7 #define LIBRARY
8 #include <unistd.h>           // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编_syscall0()等。
9
10 // 复制文件描述符函数。
11 // 下面该调用宏函数对应：int dup(int fd)。直接调用了系统中断 int 0x80，参数是__NR_dup。
12 // 其中 fd 是文件描述符。
13 _syscall1(int, dup, int, fd)
14

```

15.6 errno.c 程序

15.6.1 功能描述

该程序仅定义了一个出错号变量 `errno`。用于在函数调用失败时存放出错号。请参考 `include/errno.h` 文件。

15.6.2 代码注释

程序 15-6 linux/lib/errno.c

```

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

```

15.7 execve.c 程序

15.7.1 功能描述

运行执行程序的系统调用函数。

15.7.2 代码注释

程序 15-7 linux/lib/execve.c

```

1 /*
2  * linux/lib/execve.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #define LIBRARY

```

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

15.8 malloc.c 程序

15.8.1 功能描述

该程序中主要包括内存分配函数 `malloc()`。为了不与用户程序使用的 `malloc()` 函数相混淆，从内核 0.98 版以后就该名为 `kmalloc()`，而 `free_s()` 函数改名为 `kfree_s()`。

注意，对于应用程序使用的名称相同的内存分配函数一般在开发环境的函数库文件中实现，例如 GCC 环境中的 `libc.a` 库。由于开发环境中的库函数本身链接于用户程序中，因此它们不能直接使用内核中的 `get_free_page()` 等函数来实现内存分配函数。当然它们也没有直接管理内存页面的必要，因为只要一个进程的逻辑空间足够大，并且其数据段尾段不会覆盖位于进程逻辑地址空间末端的堆栈和环境参数区域，那么函数库 `libc.a` 中的内存分配函数只要做到按照程序动态请求的内存大小调整进程数据段末尾的设定值即可，剩下的具体内存映射等操作均由内核完成。这种调整进程数据段末端位置的操作和管理即是库中内存分配函数的主要功能，并且需要调用内核系统调用 `brk()`。参见 `kernel/sys.c` 程序第 168 行。因此若能查看开发环境中库函数实现的源代码，你将发现其中的 `malloc()`、`calloc()` 等内存分配函数除了在管理着进程动态申请的内存区域以外，最终仅调用了内核系统调用 `brk()`。开发环境中库中的内存分配函数与这里的内核库中的分配函数相同之处仅在于它们都需要对已分配内存空间进行动态管理。采用的管理方法基本是一样的。

`malloc()` 函数使用了存储桶(bucket)的原理对分配的内存进行管理。基本思想是对不同请求的内存块大小(长度)，使用存储桶目录(下面简称目录)分别进行处理。比如对于请求内存块的长度在 32 字节或 32 字节以下但大于 16 字节时，就使用存储桶目录第二项对应的存储桶描述符链表分配内存块。其基本结构示意图见图 15-1 所示。该函数目前一次所能分配的最大内存长度是一个内存页面，即 4096 字节。

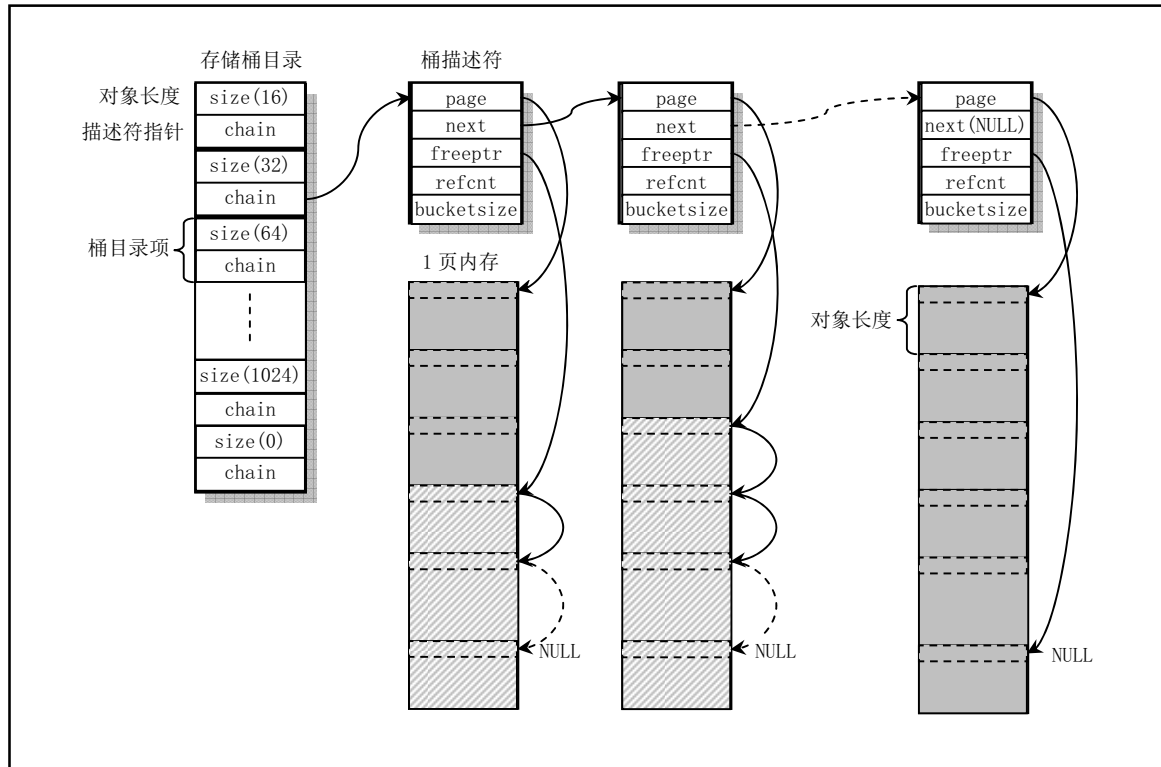


图 15-1 使用存储桶原理进行内存分配管理的结构示意图

在第一次调用 `malloc()` 函数时，首先要建立一个页面的空闲存储桶描述符(下面简称描述符)链表，其中存放着还未使用或已经使用完毕而收回的描述符。该链表结构示意图见图 15-2 所示。其中 `free_bucket_desc` 是链表头指针。从链表中取出或放入一个描述符都是从链表头开始操作。当取出一个描述符时，就将链表头指针所指向的头一个描述符取出；当释放一个空闲描述符时也是将其放在链表头处。

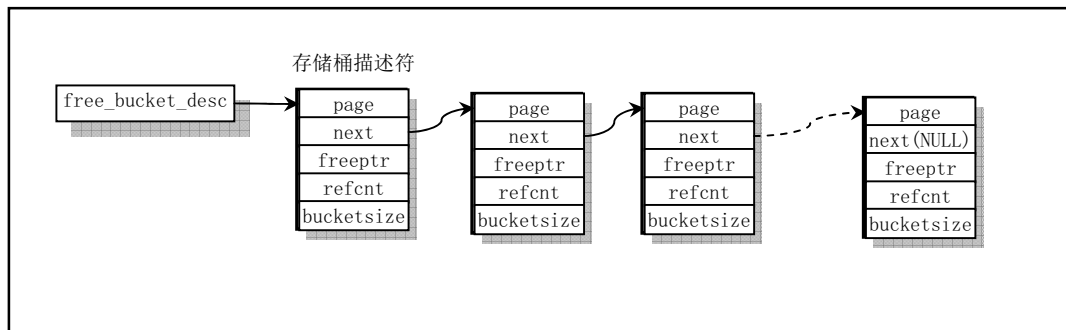


图 15-2 空闲存储桶描述符链表结构示意图

在运行过程中，如果某一时刻所有桶描述符都已占用，那么 `free_bucket_desc` 就会为 `NULL`（参见下面程序第 153 行）。因此在没有桶描述符被释放的前提下，下一次需要使用空闲桶描述符时，程序就会再次申请一个页面并在其上新建一个与上图所示相同的空闲存储桶描述符链表。

`malloc()` 函数执行的基本步骤如下：

1. 首先搜索目录，寻找适合请求内存块大小的目录项对应的描述符链表。当目录项的对象字节长度大于请求的字节长度，就算找到了相应的目录项。如果搜索完整个目录都没有找到合适的目

录项，则说明用户请求的内存块太大。

2. 在目录项对应的描述符链表中查找具有空闲空间的描述符。如果某个描述符的空闲内存指针 `freeptr` 不为 `NULL`，则表示找到了相应的描述符。如果没有找到具有空闲空间的描述符，那么我们就需要新建一个描述符。新建描述符的过程如下：
 - a. 如果空闲描述符链表头指针还是 `NULL` 的话，说明是第一次调用 `malloc()` 函数，或者所有空桶描述符都已用完。此时需要 `init_bucket_desc()` 来创建空闲描述符链表。
 - b. 然后从空闲描述符链表头处取一个描述符，初始化该描述符，令其对象引用计数为 0，对象大小等于对应目录项指定对象的长度值，并申请一内存页面，让描述符的页面指针 `page` 指向该内存页，描述符的空闲内存指针 `freeptr` 也指向页开始位置。
 - c. 对该内存页面根据本目录项所用对象长度进行页面初始化，建立所有对象的一个链表。也即每个对象的头部都存放一个指向下一个对象的指针，最后一个对象的开始处存放一个 `NULL` 指针值。
 - d. 然后将该描述符插入到对应目录项的描述符链表开始处。
3. 将该描述符的空闲内存指针 `freeptr` 复制为返回给用户的内存指针，然后调整该 `freeptr` 指向描述符对应内存页面中下一个空闲对象位置，并使该描述符引用计数值增 1。

`free_s()` 函数用于回收用户释放的内存块。基本原理是首先根据该内存块的地址换算出该内存块对应页面的地址(用页面长度进行模运算)，然后搜索目录中的所有描述符，找到对应该页面的描述符。将该释放的内存块链入 `freeptr` 所指向的空闲对象链表中，并将描述符的对象引用计数值减 1。如果引用计数值此时等于零，则表示该描述符对应的页面已经完全空出，可以释放该内存页面并将该描述符收回到空闲描述符链表中。

15.8.2 代码注释

程序 15-8 linux/lib/malloc.c

```

1 /*
2  * malloc.c --- a general purpose kernel memory allocator for Linux.
3  *
4  * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91
5  *
6  * This routine is written to be as fast as possible, so that it
7  * can be called from the interrupt level.
8  *
9  * Limitations: maximum size of memory we can allocate using this routine
10 *   is 4k, the size of a page in Linux.
11 *
12 * The general game plan is that each page (called a bucket) will only hold
13 * objects of a given size.  When all of the object on a page are released,
14 * the page can be returned to the general free pool.  When malloc() is
15 * called, it looks for the smallest bucket size which will fulfill its
16 * request, and allocate a piece of memory from that bucket pool.
17 *
18 * Each bucket has as its control block a bucket descriptor which keeps
19 * track of how many objects are in use on that page, and the free list
20 * for that page.  Like the buckets themselves, bucket descriptors are
21 * stored on pages requested from get_free_page().  However, unlike buckets,
22 * pages devoted to bucket descriptor pages are never released back to the
23 * system.  Fortunately, a system should probably only need 1 or 2 bucket
24 * descriptor pages, since a page can hold 256 bucket descriptors (which
25 * corresponds to 1 megabyte worth of bucket pages.)  If the kernel is using

```



```

26 * that much allocated memory, it's probably doing something wrong. :-)
27 *
28 * Note: malloc() and free() both call get_free_page() and free_page()
29 * in sections of code where interrupts are turned off, to allow
30 * malloc() and free() to be safely called from an interrupt routine.
31 * (We will probably need this functionality when networking code,
32 * particularly things like NFS, is added to Linux.) However, this
33 * presumes that get_free_page() and free_page() are interrupt-level
34 * safe, which they may not be once paging is added. If this is the
35 * case, we will need to modify malloc() to keep a few unused pages
36 * "pre-allocated" so that it can safely draw upon those pages if
37 * it is called from an interrupt routine.
38 *
39 * Another concern is that get_free_page() should not sleep; if it
40 * does, the code is carefully ordered so as to avoid any race
41 * conditions. The catch is that if malloc() is called re-entrantly,
42 * there is a chance that unnecessary pages will be grabbed from the
43 * system. Except for the pages for the bucket descriptor page, the
44 * extra pages will eventually get released back to the system, though,
45 * so it isn't all that bad.
46 */
47
/*
 * malloc.c - Linux 的通用内核内存分配函数。
 *
 * 由 Theodore Ts'o 编制 (tytso@mit.edu), 11/29/91
 *
 * 该函数被编写成尽可能地快, 从而可以从中断层调用此函数。
 *
 * 限制: 使用该函数一次所能分配的最大内存是 4k, 也即 Linux 中内存页面的大小。
 *
 * 编写该函数所遵循的一般规则是每页(被称为一个存储桶)仅分配所要容纳对象的大小。
 * 当一页上的所有对象都释放后, 该页就可以返回通用空闲内存池。当 malloc() 被调用
 * 时, 它会寻找满足要求的最小的存储桶, 并从该存储桶中分配一块内存。
 *
 * 每个存储桶都有一个作为其控制用的存储桶描述符, 其中记录了页面上有多少对象正被
 * 使用以及该页上空闲内存的列表。就象存储桶自身一样, 存储桶描述符也是存储在使
 * 用 get_free_page() 申请到的页面上的, 但是与存储桶不同的是, 桶描述符所占用的页面
 * 将不再会释放给系统。幸运的是一个系统大约只需要 1 到 2 页的桶描述符页面, 因为一
 * 个页面可以存放 256 个桶描述符(对应 1MB 内存的存储桶页面)。如果系统为桶描述符分
 * 配了许多内存, 那么肯定系统什么地方出了问题☺。
 *
 * 注意! malloc() 和 free() 两者关闭了中断的代码部分都调用了 get_free_page() 和
 * free_page() 函数, 以使 malloc() 和 free() 可以安全地被从中断程序中调用
 * (当网络代码, 尤其是 NFS 等被加入到 Linux 中时就需要这种功能)。但前
 * 提是假设 get_free_page() 和 free_page() 是可以安全地在中断级程序中使用的,
 * 这在一旦加入了分页处理之后就很可能不是安全的。如果真是这种情况, 那么我
 * 们就需要修改 malloc() 来“预先分配”几页不用的内存, 如果 malloc() 和 free()
 * 被从中断程序中调用时就可以安全地使用这些页面。
 *
 * 另外需要考虑到的是 get_free_page() 不应该睡眠; 如果会睡眠的话, 则为了防止
 * 任何竞争条件, 代码需要仔细地安排顺序。关键在于如果 malloc() 是可以重入地
 * 被调用的话, 那么就会存在不必要的页面被从系统中取走的机会。除了用于桶描述

```

```

*      符的页面，这些额外的页面最终会释放给系统，所以并不是象想象的那样不好。
*/

48 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
49 #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。
50 #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。
51
// 存储桶描述符结构。
52 struct bucket_desc {      /* 16 bytes */
53     void *page;           // 该桶描述符对应的内存页面指针。
54     struct bucket_desc *next; // 下一个描述符指针。
55     void *freeptr;        // 指向本桶中空闲内存位置的指针。
56     unsigned short refcnt; // 引用计数。
57     unsigned short bucket_size; // 本描述符对应存储桶的大小。
58 };
59
// 存储桶描述符目录结构。
60 struct bucket_dir {       /* 8 bytes */
61     int size;             // 该存储桶的大小(字节数)。
62     struct bucket_desc *chain; // 该存储桶目录项的桶描述符链表指针。
63 };
64
65 /*
66  * The following is the where we store a pointer to the first bucket
67  * descriptor for a given size.
68  *
69  * If it turns out that the Linux kernel allocates a lot of objects of a
70  * specific size, then we may want to add that specific size to this list,
71  * since that will allow the memory to be allocated more efficiently.
72  * However, since an entire page must be dedicated to each specific size
73  * on this list, some amount of temperance must be exercised here.
74  *
75  * Note that this list must be kept in order.
76  */
77 /*
78  * 下面是我们存放第一个给定大小存储桶描述符指针的地方。
79  *
80  * 如果 Linux 内核分配了许多指定大小的对象，那么我们就希望将该指定的大小加到
81  * 该列表(链表)中，因为这样可以使内存的分配更有效。但是，因为一页完整内存页面
82  * 必须用于列表中指定大小的所有对象，所以需要总数方面的测试操作。
83  */
// 存储桶目录列表(数组)。
77 struct bucket_dir bucket_dir[] = {
78     { 16, (struct bucket_desc *) 0}, // 16 字节长度的内存块。
79     { 32, (struct bucket_desc *) 0}, // 32 字节长度的内存块。
80     { 64, (struct bucket_desc *) 0}, // 64 字节长度的内存块。
81     {128, (struct bucket_desc *) 0}, // 128 字节长度的内存块。
82     {256, (struct bucket_desc *) 0}, // 256 字节长度的内存块。
83     {512, (struct bucket_desc *) 0}, // 512 字节长度的内存块。
84     {1024, (struct bucket_desc *) 0}, // 1024 字节长度的内存块。
85     {2048, (struct bucket_desc *) 0}, // 2048 字节长度的内存块。
86     {4096, (struct bucket_desc *) 0}, // 4096 字节(1 页)内存。
87     { 0, (struct bucket_desc *) 0}}; /* End of list marker */

```

```

88
89 /*
90  * This contains a linked list of free bucket descriptor blocks
91  */
92 /*
93  * 下面是含有空闲桶描述符内存块的链表。
94  */
95 struct bucket_desc *free_bucket_desc = (struct bucket_desc *) 0;
96
97 /*
98  * This routine initializes a bucket description page.
99  */
100 /*
101  * 下面的子程序用于初始化一页桶描述符页面。
102  */
103 // 建立空闲桶描述符链表，并让 free_bucket_desc 指向第一个空闲桶描述符。
104 static inline void init_bucket_desc()
105 {
106     struct bucket_desc *bdesc, *first;
107     int i;
108
109     // 申请一页内存，用于存放桶描述符。如果失败，则显示初始化桶描述符时内存不够出错信息，死机。
110     first = bdesc = (struct bucket_desc *) get_free_page();
111     if (!bdesc)
112         panic("Out of memory in init_bucket_desc()");
113     // 首先计算一页内存中可存放的桶描述符数量，然后对其建立单向连接指针。
114     for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {
115         bdesc->next = bdesc+1;
116         bdesc++;
117     }
118     /*
119     * This is done last, to avoid race conditions in case
120     * get_free_page() sleeps and this routine gets called again...
121     */
122     /*
123     * 这是在最后处理的，目的是为了避免在 get_free_page() 睡眠时该子程序又被
124     * 调用而引起的竞争条件。
125     */
126     // 将空闲桶描述符指针 free_bucket_desc 加入链表中。
127     bdesc->next = free_bucket_desc;
128     free_bucket_desc = first;
129 }
130
131 // 分配动态内存函数。
132 // 参数: len - 请求的内存块长度。
133 // 返回: 指向被分配内存的指针。如果失败则返回 NULL。
134 void *malloc(unsigned int len)
135 {
136     struct bucket_dir *bdir;
137     struct bucket_desc *bdesc;
138     void *retval;
139 }

```

```

123     /*
124     * First we search the bucket_dir to find the right bucket change
125     * for this request.
126     */
127     /*
128     * 首先我们搜索存储桶目录 bucket_dir 来寻找适合请求的桶大小。
129     */
130     // 搜索存储桶目录，寻找适合申请内存块大小的桶描述符链表。如果目录项的桶字节数大于请求的字节
131     // 数，就找到了对应的桶目录项。
132     for (bdir = bucket_dir; bdir->size; bdir++)
133         if (bdir->size >= len)
134             break;
135     // 如果搜索完整个目录都没有找到合适大小的目录项，则表明所请求的内存块大小太大，超出了该
136     // 程序的分配限制(最长为 1 个页面)。于是显示出错信息，死机。
137     if (!bdir->size) {
138         printk("malloc called with impossibly large argument (%d)\n",
139               len);
140         panic("malloc: bad arg");
141     }
142     /*
143     * Now we search for a bucket descriptor which has free space
144     */
145     /*
146     * 现在我们来搜索具有空闲空间的桶描述符。
147     */
148     cli(); /* Avoid race conditions */ /* 为了避免出现竞争条件，首先关中断 */
149     // 搜索对应桶目录项中描述符链表，查找具有空闲空间的桶描述符。如果桶描述符的空闲内存指针
150     // freeptr 不为空，则表示找到了相应的桶描述符。
151     for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
152         if (bdesc->freeptr)
153             break;
154     /*
155     * If we didn't find a bucket with free space, then we'll
156     * allocate a new one.
157     */
158     /*
159     * 如果没有找到具有空闲空间的桶描述符，那么我们就需要新建一个该目录项的描述符。
160     */
161     if (!bdesc) {
162         char *cp;
163         int i;
164         // 若 free_bucket_desc 还为空时，表示第一次调用该程序，或者链表中所有空桶描述符都已用完。
165         // 此时就需要申请一个页面并在其上建立并初始化空闲描述符链表。free_bucket_desc 会指向第一
166         // 个空闲桶描述符。
167         if (!free_bucket_desc)
168             init_bucket_desc();
169         // 取 free_bucket_desc 指向的空闲桶描述符，并让 free_bucket_desc 指向下一个空闲桶描述符。
170         bdesc = free_bucket_desc;
171         free_bucket_desc = bdesc->next;
172         // 初始化该新的桶描述符。令其引用数量等于 0；桶的大小等于对应桶目录的大小；申请一内存页面，
173         // 让描述符的页面指针 page 指向该页面；空闲内存指针也指向该页开头，因为此时全为空闲。
174         bdesc->refcnt = 0;

```

```

155         bdesc->bucket_size = bdir->size;
156         bdesc->page = bdesc->freeptr = (void *) cp = get\_free\_page();
// 如果申请内存页面操作失败，则显示出错信息，死机。
157         if (!cp)
158             panic("Out of memory in kernel malloc()");
159         /* Set up the chain of free objects */
        /* 在该页空闲内存中建立空闲对象链表 */
// 以该桶目录项指定的桶大小为对象长度，对该页内存进行划分，并使每个对象的开始 4 字节设置
// 成指向下一对象的指针。
160         for (i=PAGE\_SIZE/bdir->size; i > 1; i--) {
161             *((char **) cp) = cp + bdir->size;
162             cp += bdir->size;
163         }
// 最后一个对象开始处的指针设置为 0(NULL)。
// 然后让该桶描述符的下一描述符指针字段指向对应桶目录项指针 chain 所指的描述符，而桶目录的
// chain 指向该桶描述符，也即将该描述符插入到描述符链链头处。
164         *((char **) cp) = 0;
165         bdesc->next = bdir->chain; /* OK, link it in! */ /* OK, 将其链入! */
166         bdir->chain = bdesc;
167     }
// 返回指针即等于该描述符对应页面的当前空闲指针。然后调整该空闲空间指针指向下一个空闲对象，
// 并使描述符中对应页面中对象引用计数增 1。
168     retval = (void *) bdesc->freeptr;
169     bdesc->freeptr = *((void **) retval);
170     bdesc->refcnt++;
// 最后开放中断，并返回指向空闲内存对象的指针。
171     sti(); /* OK, we're safe again */ /* OK, 现在我们又安全了 */
172     return(retval);
173 }
174
175 /*
176  * Here is the free routine. If you know the size of the object that you
177  * are freeing, then free_s() will use that information to speed up the
178  * search for the bucket descriptor.
179  *
180  * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
181  */
/*
    * 下面是释放子程序。如果你知道释放对象的大小，则 free_s() 将使用该信息加速
    * 搜寻对应桶描述符的速度。
    *
    * 我们将定义一个宏，使得 "free(x)" 成为 "free_s(x, 0)"。
    */
//// 释放存储桶对象。
// 参数: obj - 对应对象指针; size - 大小。
182 void free\_s(void *obj, int size)
183 {
184     void *page;
185     struct bucket\_dir *bdir;
186     struct bucket\_desc *bdesc, *prev;
187
188     /* Calculate what page this object lives in */
    /* 计算该对象所在的页面 */

```

```

189     page = (void *) ((unsigned long) obj & 0xfffff000);
190     /* Now search the buckets looking for that page */
    /* 现在搜索存储桶目录项所链接的桶描述符，寻找该页面 */
    //
191     for (bdir = bucket\_dir; bdir->size; bdir++) {
192         prev = 0;
193         /* If size is zero then this conditional is always false */
        /* 如果参数 size 是 0，则下面条件肯定是 false */
194         if (bdir->size < size)
195             continue;
        // 搜索对应目录项中链接的所有描述符，查找对应页面。如果某描述符页面指针等于 page 则表示找到
        // 了相应的描述符，跳转到 found。如果描述符不含有对应 page，则让描述符指针 prev 指向该描述符。
196         for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
197             if (bdesc->page == page)
198                 goto found;
199             prev = bdesc;
200         }
201     }
    // 若搜索了对应目录项的所有描述符都没有找到指定的页面，则显示出错信息，死机。
202     panic("Bad address passed to kernel free_s()");
203 found:
    // 找到对应的桶描述符后，首先关中断。然后将该对象内存块链入空闲块对象链表中，并使该描述符
    // 的对象引用计数减 1。
204     cli(); /* To avoid race conditions */ /* 为了避免竞争条件 */
205     *((void **)obj) = bdesc->freeptr;
206     bdesc->freeptr = obj;
207     bdesc->refcnt--;
    // 如果引用计数已等于 0，则我们就可以释放对应的内存页面和该桶描述符。
208     if (bdesc->refcnt == 0) {
209         /*
210         * We need to make sure that prev is still accurate. It
211         * may not be, if someone rudely interrupted us....
212         */
        /*
        * 我们需要确信 prev 仍然是正确的，若某程序粗鲁地中断了我们
        * 就有可能不是了。
        */
        // 如果 prev 已经不是搜索到的描述符的前一个描述符，则重新搜索当前描述符的前一个描述符。
213         if ((prev && (prev->next != bdesc)) ||
214             (!prev && (bdir->chain != bdesc)))
215             for (prev = bdir->chain; prev; prev = prev->next)
216                 if (prev->next == bdesc)
217                     break;
        // 如果找到该前一个描述符，则从描述符链中删除当前描述符。
218         if (prev)
219             prev->next = bdesc->next;
        // 如果 prev==NULL，则说明当前一个描述符是该目录项首个描述符，也即目录项中 chain 应该直接
        // 指向当前描述符 bdesc，否则表示链表有问题，则显示出错信息，死机。因此，为了将当前描述符
        // 从链表中删除，应该让 chain 指向下一个描述符。
220         else {
221             if (bdir->chain != bdesc)
222                 panic("malloc bucket chains corrupted");
223             bdir->chain = bdesc->next;

```

```

224         }
    // 释放当前描述符所操作的内存页面，并将该描述符插入空闲描述符链表开始处。
225         free_page((unsigned long) bdesc->page);
226         bdesc->next = free_bucket_desc;
227         free_bucket_desc = bdesc;
228     }
    // 开中断，返回。
229     sti();
230     return;
231 }
232
233

```

15.9 open.c 程序

15.9.1 功能描述

`open()` 系统调用用于将一个文件名转换成一个文件描述符。当调用成功时，返回的文件描述符将是进程没有打开的最小数值的描述符。该调用创建一个新的打开文件，并不与任何其他进程共享。在执行 `exec` 函数时，该新的文件描述符将始终保持着打开状态。文件的读写指针被设置在文件开始位置。

参数 `flag` 是 `O_RDONLY`、`O_WRONLY`、`O_RDWR` 之一，分别代表文件只读打开、只写打开和读写打开方式，可以与其他一些标志一起使用。(参见 `fs/open.c`, 138 行)

15.9.2 代码注释

程序 15-9 linux/lib/open.c

```

1  /*
2   *  linux/lib/open.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #define  _LIBRARY
8  #include <unistd.h>    // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                        // 如定义了 __LIBRARY__，则还含系统调用号和内嵌汇编 _syscall0() 等。
9  #include <stdarg.h>    // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
                        // 类型 (va_list) 和三个宏 (va_start, va_arg 和 va_end)，用于
                        // vsprintf、vprintf、vfprintf 函数。
10
    // 打开文件函数。
    // 打开并有可能创建一个文件。
    // 参数: filename - 文件名; flag - 文件打开标志; ...
    // 返回: 文件描述符，若出错则置出错码，并返回 -1。
    // 第 13 行定义了一个寄存器变量 res，该变量将被保存在一个寄存器中，以便于高效访问和操作。
    // 若想指定存放的寄存器（例如 eax），那么可以把该句写成 “register int res asm("ax");”。
11 int open(const char * filename, int flag, ...)

```

```

12 {
13     register int res;
14     va_list arg;
15     // 利用 va_start() 宏函数, 取得 flag 后面参数的指针, 然后调用系统中断 int 0x80, 功能 open 进行
    // 文件打开操作。
    // %0 - eax(返回的描述符或出错码); %1 - eax(系统中断调用功能号 __NR_open);
    // %2 - ebx(文件名 filename); %3 - ecx(打开文件标志 flag); %4 - edx(后随参数文件属性 mode)。
16     va_start(arg, flag);
17     __asm__ ("int $0x80"
18             : "=a" (res)
19             : "" (__NR_open), "b" (filename), "c" (flag),
20             "d" (va_arg(arg, int)));
    // 系统中断调用返回值大于或等于 0, 表示是一个文件描述符, 则直接返回之。
21     if (res >= 0)
22         return res;
    // 否则说明返回值小于 0, 则代表一个出错码。设置该出错码并返回-1。
23     errno = -res;
24     return -1;
25 }
26

```

15.10 setsid.c 程序

15.10.1 功能描述

该程序包括一个 `setsid()` 系统调用函数。如果调用的进程不是一个组的领导时, 该函数用于创建一个新会话。则调用进程将成为该新会话的领导、新进程组的组领导, 并且没有控制终端。调用进程的组 id 和会话 id 被设置成进程的 PID(进程标识符)。调用进程将成为新进程组和新会话中的唯一进程。

15.10.2 代码注释

程序 15-10 linux/lib/setsid.c

```

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

```

15.11 string.c 程序

15.11.1 功能描述

所有字符串操作函数已经存在于 `string.h` 中，这里通过首先声明 `'extern'` 和 `'inline'` 前缀为空，然后再包含 `string.h` 头文件，实现了 `string.c` 中仅包含字符串函数的实现代码。参见 `include/string.h` 头文件前的说明。

15.11.2 代码注释

程序 15-11 `linux/lib/string.c`

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

15.12 wait.c 程序

15.12.1 功能描述

该程序包括函数 `waitpid()` 和 `wait()`。这两个函数允许进程获取与其子进程之一的状态信息。各种选项允许获取已经终止或停止的子进程状态信息。如果存在两个或两个以上子进程的状态信息，则报告顺序是不指定的。

`wait()` 将挂起当前进程，直到其子进程之一退出（终止），或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。

`waitpid()` 挂起当前进程，直到 `pid` 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。

如果 `pid = -1`，`options = 0`，则 `waitpid()` 的作用与 `wait()` 函数一样。否则其行为将随 `pid` 和 `options` 参数的不同而不同。（参见 `kernel/exit.c, 142`）

9`//// 写文件系统调用函数。``// 该宏结构对应于函数: int write(int fd, const char * buf, off_t count)``// 参数: fd - 文件描述符; buf - 写缓冲区指针; count - 写字节数。``// 返回: 成功时返回写入的字节数(0 表示写入 0 字节); 出错时将返回-1, 并且设置了出错号。`10 `_syscall13(int, write, int, fd, const char *, buf, off_t, count)`11

第16章 建造工具(tools)

Linux 内核源代码中的 `tools` 目录中包含一个生成内核磁盘映像文件的工具程序 `build.c`，该程序将单独编译成可执行文件，在 `linux/` 目录下的 `Makefile` 文件中被调用运行，用于将所有内核编译代码连接和合并成一个可运行的内核映像文件 `Image`。具体方法是对 `boot/` 中的 `bootsect.s`、`setup.s` 使用 8086 汇编器进行编译，分别生成各自的执行模块。再对源代码中的其他所有程序使用 GNU 的编译器 `gcc/gas` 进行编译，并连接成模块 `system`。然后使用 `build` 工具将这三块组合成一个内核映像文件 `Image`。基本编译连接/组合结构如图 16-1 所示。

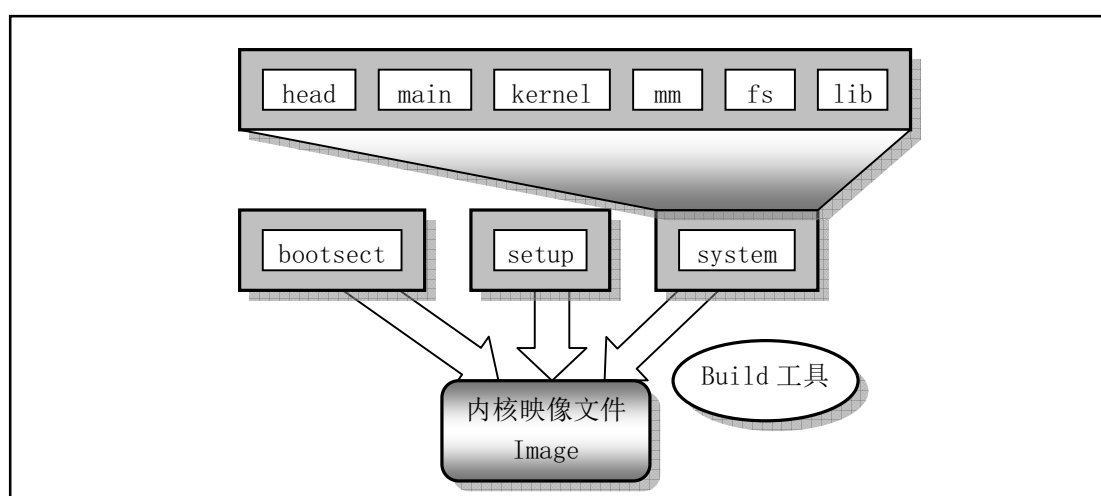


图 16-1 内核编译连接/组合结构

16.1 build.c 程序

16.1.1 功能概述

在 `linux/Makefile` 文件第 42 行上，执行 `build` 程序的命令行形式如下所示：

```
tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) > Image
```

`build` 程序使用 4 个参数，分别是 `bootsect` 文件名、`setup` 文件名、`system` 文件名和可选的根文件系统设备文件名。`bootsect` 和 `setup` 程序是由 `as86` 和 `ld86` 编译链接产生，它们具有 MINIX 执行文件格式（参见程序列表后的说明），而 `system` 模块是由源代码各个子目录中编译产生的模块链接而成，具有 GNU `a.out` 执行文件格式。`build` 程序的主要工作就是去掉 `bootsect` 和 `setup` 的 MINIX 执行文件头结构信息、去掉 `system` 文件中的 `a.out` 头结构信息，只保留它们的代码和数据部分，然后把它们顺序组合在一起写入名为 `Image` 的文件中。

程序首先检查命令行上最后一个根设备文件名可选参数，若其存在，则读取该设备文件的状态信息结构（`stat`），取出设备号。若命令行上不带该参数，则使用默认值。

然后对 bootsect 文件进行处理，读取该文件的 minix 执行头部信息，判断其有效性，然后读取随后 512 字节的引导代码数据，判断其是否具有可引导标志 0xAA55，并将前面获取的根设备号写入到 508,509 位处，最后将该 512 字节代码数据写到 stdout 标准输出，由 Make 文件重定向到 Image 文件。

接下来以类似的方法处理 setup 文件。若该文件长度小于 4 个扇区，则用 0 将其填满为 4 个扇区的长度，并写到标准输出 stdout 中。

最后处理 system 文件。该文件是使用 GCC 编译器产生，所以其执行头部格式是 GCC 类型的，与 linux 定义的 a.out 格式一样。在判断执行入口点是 0 后，就将数据写到标准输出 stdout 中。若其代码数据长度超过 128KB，则显示出错信息。最终形成的内核 Image 文件格式是：

- 第 1 个扇区上存放的是 bootsect 代码，长度正好 512 字节；
- 从第 2 个扇区开始的 4 个扇区（2 - 5 扇区）存放着 setup 代码，长度不超过 4 个扇区大小；
- 从第 6 个扇区开始存放 system 模块的代码，其长度不超过 build.c 第 35 行上定义的大小。

16.1.2 代码注释

程序 16-1 linux/tools/build.c

```
1 /*
2  * linux/tools/build.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * This file builds a disk-image from three different files:
9  *
10 * - bootsect: max 510 bytes of 8086 machine code, loads the rest
11 * - setup: max 4 sectors of 8086 machine code, sets up system parm
12 * - system: 80386 code for actual system
13 *
14 * It does some checking that all files are of the correct type, and
15 * just writes the result to stdout, removing headers and padding to
16 * the right amount. It also writes some system data to stderr.
17 */
18 /*
19  * 该程序从三个不同的程序中创建磁盘映像文件：
20  *
21 * - bootsect: 该文件的 8086 机器码最长为 510 字节，用于加载其他程序。
22 * - setup: 该文件的 8086 机器码最长为 4 个磁盘扇区，用于设置系统参数。
23 * - system: 实际系统的 80386 代码。
24 *
25 * 该程序首先检查所有程序模块的类型是否正确，并将检查结果在终端上显示出来，
26 * 然后删除模块头部并扩充大正确的长度。该程序也会将一些系统数据写到 stderr。
27 */
28 /*
29  * Changes by tytso to allow root device specification
30 */
31 /*
32  * tytso 对该程序作了修改，以允许指定根文件设备。
33 */
34
```

```

23 #include <stdio.h>          /* fprintf */          /* 使用其中的 fprintf() */
24 #include <string.h>         /* 字符串操作 */
25 #include <stdlib.h>         /* contains exit */     /* 含有 exit() */
26 #include <sys/types.h>      /* unistd.h needs this */ /* 供 unistd.h 使用 */
27 #include <sys/stat.h>       /* 文件状态信息结构 */
28 #include <linux/fs.h>       /* 文件系统 */
29 #include <unistd.h>         /* contains read/write */ /* 含有 read()/write() */
30 #include <fcntl.h>          /* 文件操作模式符号常数 */
31
32 #define MINIX_HEADER 32      // minix 二进制目标文件模块头部长度为 32 字节。
33 #define GCC_HEADER 1024     // GCC 头部信息长度为 1024 字节。
34
35 #define SYS_SIZE 0x2000     // system 文件最长节数(字节数为 SYS_SIZE*16=128KB)。
36
// 默认地把 Linux 根文件系统所在设备设置为在第 2 个硬盘的第 1 个分区上（即设备号为 0x0306），
// 是因为 Linus 当时开发 Linux 时，把第 1 个硬盘用作 MINIX 系统盘，而第 2 个硬盘用作 Linux
// 的根文件系统盘。
37 #define DEFAULT_MAJOR_ROOT 3 // 默认根设备主设备号 - 3（硬盘）。
38 #define DEFAULT_MINOR_ROOT 6 // 默认根设备次设备号 - 6（第 2 个硬盘的第 1 分区）。
39
40 /* max nr of sectors of setup: don't change unless you also change
41  * bootsect etc */
// 下面指定 setup 模块占的最大扇区数：不要改变该值，除非也改变 bootsect 等相应文件。
42 #define SETUP_SECTS 4       // setup 最大长度为 4 个扇区（4*512 字节）。
43
44 #define STRINGIFY(x) #x      // 把 x 转换成字符串类型，用于出错显示语句中。
45
//// 显示出错信息，并终止程序。
46 void die(char * str)
47 {
48     fprintf(stderr, "%s\n", str);
49     exit(1);
50 }
51
// 显示程序使用方法，并退出。
52 void usage(void)
53 {
54     die("Usage: build bootsect setup system [rootdev] [> image]^");
55 }
56
57 int main(int argc, char ** argv)
58 {
59     int i, c, id;
60     char buf[1024];
61     char major_root, minor_root;
62     struct stat sb;
63
// 如果程序命令行参数不是 4 或 5 个（执行程序名本身算作其中 1 个），则显示程序用法并退出。
64     if ((argc != 4) && (argc != 5))
65         usage();
// 如果参数是 5 个，则说明带有根设备名。
66     if (argc == 5) {
// 如果根设备名不是软盘("FLOPPY")，则取该设备文件的状态信息。若出错则显示信息并退出，

```

```

// 否则取该设备名状态结构中的主设备号和次设备号。如果根设备就是 FLOPPY 设备，则让主设备
// 号和次设备号取 0。表示根设备就是当前启动引导设备。
67         if (strcmp(argv[4], "FLOPPY")) {
68             if (stat(argv[4], &sb)) {
69                 perror(argv[4]);
70                 die("Couldn't stat root device.");
71             }
72             major_root = MAJOR(sb.st_rdev); // 取设备名状态结构中设备号。
73             minor_root = MINOR(sb.st_rdev);
74         } else {
75             major_root = 0;
76             minor_root = 0;
77         }
// 若参数只有 4 个，则让主设备号和次设备号等于系统默认的根设备号。
78     } else {
79         major_root = DEFAULT_MAJOR_ROOT;
80         minor_root = DEFAULT_MINOR_ROOT;
81     }

// 接下来在标准错误终端上显示所选择的根设备主、次设备号。如果主设备号不等于 2（软盘）或
// 3（硬盘），也不等于 0（取系统默认根设备），则显示出错信息并退出。
82     fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
83     if ((major_root != 2) && (major_root != 3) &&
84         (major_root != 0)) {
85         fprintf(stderr, "Illegal root device (major = %d)\n",
86             major_root);
87         die("Bad root device --- major #");
88     }
// 初始化 buf 缓冲区，全置 0。
89     for (i=0; i<sizeof buf; i++) buf[i]=0;
// 以只读方式打开参数 1 指定的文件 (bootsect)，若出错则显示出错信息，退出。
90     if ((id=open(argv[1], O_RDONLY, 0))<0)
91         die("Unable to open 'boot'");
// 读取文件中的 minix 执行头部信息 (参见列表后说明)，若出错则显示出错信息，退出。
92     if (read(id, buf, MINIX_HEADER) != MINIX_HEADER)
93         die("Unable to read header of 'boot'");
// 0x0301 - MINIX 头部 a_magic 魔数；0x10 - a_flag 可执行；0x04 - a_cpu，Intel 8086 机器码。
94     if (((long *) buf)[0]!=0x04100301)
95         die("Non-Minix header of 'boot'");
// 判断头部长度字段 a_hdrlen（字节）是否正确（32 字节）。（后三字节正好没有用，是 0）
96     if (((long *) buf)[1]!=MINIX_HEADER)
97         die("Non-Minix header of 'boot'");
// 判断数据段长 a_data 字段(long)内容是否为 0。
98     if (((long *) buf)[3]!=0)
99         die("Illegal data segment in 'boot'");
// 判断堆 a_bss 字段(long)内容是否为 0。
100    if (((long *) buf)[4]!=0)
101        die("Illegal bss in 'boot'");
// 判断执行点 a_entry 字段(long)内容是否为 0。
102    if (((long *) buf)[5] != 0)
103        die("Non-Minix header of 'boot'");
// 判断符号表长字段 a_sym 的内容是否为 0。
104    if (((long *) buf)[7] != 0)

```

```

105         die("Illegal symbol table in 'boot'");
// 读取实际代码数据, 应该返回读取字节数为 512 字节。
106         i=read(id,buf,sizeof buf);
107         fprintf(stderr, "Boot sector %d bytes. \n",i);
108         if (i != 512)
109             die("Boot block must be exactly 512 bytes");
// 判断 boot 块 0x510 处是否有可引导标志 0xAA55。
110         if ((*unsigned short *) (buf+510)) != 0xAA55)
111             die("Boot block hasn't got boot flag (0xAA55)");
// 引导块的 508, 509 偏移处存放的是根设备号。
112         buf[508] = (char) minor_root;
113         buf[509] = (char) major_root;
// 将该 boot 块 512 字节的数据写到标准输出 stdout, 若写出字节数不对, 则显示出错信息, 退出。
114         i=write(1,buf,512);
115         if (i!=512)
116             die("Write call failed");
// 最后关闭 bootsect 模块文件。
117         close (id);
118
// 现在开始处理 setup 模块。首先以只读方式打开该模块, 若出错则显示出错信息, 退出。
119         if ((id=open(argv[2], O_RDONLY, 0))<0)
120             die("Unable to open 'setup'");
// 读取该文件中的 minix 执行头部信息 (32 字节), 若出错则显示出错信息, 退出。
121         if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
122             die("Unable to read header of 'setup'");
// 0x0301 - minix 头部 a_magic 魔数; 0x10 - a_flag 可执行; 0x04 - a_cpu, Intel 8086 机器码。
123         if (((long *) buf)[0]!=0x04100301)
124             die("Non-Minix header of 'setup'");
// 判断头部长度字段 a_hdrlen (字节) 是否正确。(后三字节正好没有用, 是 0)
125         if (((long *) buf)[1]!=MINIX_HEADER)
126             die("Non-Minix header of 'setup'");
// 判断数据段长 a_data 字段(long)内容是否为 0。
127         if (((long *) buf)[3]!=0)
128             die("Illegal data segment in 'setup'");
// 判断堆 a_bss 字段(long)内容是否为 0。
129         if (((long *) buf)[4]!=0)
130             die("Illegal bss in 'setup'");
// 判断执行点 a_entry 字段(long)内容是否为 0。
131         if (((long *) buf)[5] != 0)
132             die("Non-Minix header of 'setup'");
// 判断符号表长字段 a_sym 的内容是否为 0。
133         if (((long *) buf)[7] != 0)
134             die("Illegal symbol table in 'setup'");
// 读取随后的执行代码数据, 并写到标准输出 stdout。
135         for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
136             if (write(1,buf,c)!=c)
137                 die("Write call failed");
//关闭 setup 模块文件。
138         close (id);
// 若 setup 模块长度大于 4 个扇区, 则算出错, 显示出错信息, 退出。
139         if (i > SETUP_SECTS*512)
140             die("Setup exceeds " STRINGIFY(SETUP_SECTS)
141                 " sectors - rewrite build/boot/setup");

```

```

// 在标准错误 stderr 显示 setup 文件的长度值。
142     fprintf(stderr, "Setup is %d bytes. \n", i);
// 将缓冲区 buf 清零。
143     for (c=0 ; c<sizeof(buf) ; c++)
144         buf[c] = '\0';
// 若 setup 长度小于 4*512 字节，则用\0 将 setup 补足为 4*512 字节。
145     while (i<SETUP_SECTS*512) {
146         c = SETUP_SECTS*512-i;
147         if (c > sizeof(buf))
148             c = sizeof(buf);
149         if (write(1, buf, c) != c)
150             die("Write call failed");
151         i += c;
152     }
153
// 下面处理 system 模块。首先以只读方式打开该文件。
154     if ((id=open(argv[3], O_RDONLY, 0))<0)
155         die("Unable to open 'system'");
// system 模块是 GCC 格式的文件，先读取 GCC 格式的头部结构信息(linux 的执行文件也采用该格式)。
156     if (read(id, buf, GCC_HEADER) != GCC_HEADER)
157         die("Unable to read header of 'system'");
// 该结构中的执行代码入口点字段 a_entry 值应为 0。
158     if (((long *) buf)[5] != 0)
159         die("Non-GCC header of 'system'");
// 读取随后的执行代码数据，并写到标准输出 stdout。
160     for (i=0 ; (c=read(id, buf, sizeof buf))>0 ; i+=c )
161         if (write(1, buf, c)!=c)
162             die("Write call failed");
// 关闭 system 文件，并向 stderr 上打印 system 的字节数。
163     close(id);
164     fprintf(stderr, "System is %d bytes. \n", i);
// 若 system 代码数据长度超过 SYS_SIZE 节（或 128KB 字节），则显示出错信息，退出。
165     if (i > SYS_SIZE*16)
166         die("System is too big");
167     return(0);
168 }
169

```

16.1.3 相关信息

16.1.3.1 可执行文件头部数据结构

Minix 可执行文件 a.out 的头部结构如下所示（参见 minix 2.0 源代码 01400 行开始）：

```

struct exec {
    unsigned char a_magic[2];    // 执行文件魔数。
    unsigned char a_flags;      // 标志（参见下面说明）。
    unsigned char a_cpu;        // cpu 标识号。
    unsigned char a_hdrlen;     // 保留头部长度，32 字节或 48 字节。
    unsigned char a_unused;     // 保留给将来使用。
    unsigned short a_version;    // 版本信息（目前未用）。
    long          a_text;       // 代码段长度，字节数。

```

```
long      a_data;          // 数据段长度，字节数。
long      a_bss;           // 堆长度，字节数。
long      a_entry;        // 执行入口点地址。
long      a_total;        // 分配的内存总量。
long      a_syms;          // 符号表大小。
    // 若头部为 32 字节，就到此为止。
long      a_trsize;        // 代码段重定位表长度。
long      a_drsiz;        // 数据段重定位表长度。
long      a_tbase;        // 代码段重定位基址。
long      a_dbase;        // 数据段重定位基址。
};
```

其中，MINIX 执行文件的魔数字段 `a_magic[]` 值为：

```
a_magic[0] = 0x01
a_magic[1] = 0x03
```

标志字段 `a_flags` 定义为：

```
A_UZP    0x01            // 未映射的 0 页（页数）。
A_PAL    0x02            // 以页边界调整的可执行文件。
A_NSYM   0x04            // 新型符号表。
A_EXEC   0x10            // 可执行文件。
A_SEP    0x20            // 代码和数据是分开的（I 和 D 独立）。
```

CPU 标识号字段 `a_cpu` 为：

```
A_NONE   0x00            // 未知。
A_I8086   0x04            // Intel i8086/8088。
A_M68K    0x0B            // Motorola m68000。
A_NS16K   0x0C            // 国家半导体公司 16032。
A_I80386  0x10            // Intel i80386。
A_SPARC   0x17            // Sun 公司 SPARC。
```

MINIX 执行头结构 `exec` 与 Linux 0.11 系统所使用的 `a.out` 格式执行文件头结构类似。Linux `a.out` 格式执行文件的头部结构及相关信息请参见 `linux/include/a.out.h` 文件。

第17章 实验环境设置与使用方法

为了配合 Linux 0.1x (0.11 或 0.12) 内核工作原理的学习, 本章介绍利用 PC 机仿真软件 and 在实际计算机上运行 Linux 0.1x 系统的方法。其中包括内核编译过程、PC 仿真环境下文件的访问和复制、引导盘和根文件系统的制作方法以及 Linux 0.1x 系统的使用方法等。最后还说明了如何对内核代码作少量语法修改, 使其在现有的 RedHat 9 系统 (gcc 3.x) 下能顺利通过编译, 并制作出内核映像文件。

在开始进行实验之前, 首先准备好一些有用的工具软件。若在 Windows 平台上进行实验学习, 我们需要准备好以下几个软件:

- Bochs 2.2.x 开放源代码的 PC 机仿真软件包。
- UltraEdit 超级编辑器。可用来编辑二进制文件。
- WinImage DOS 格式软盘映像文件的读写软件。

若在现代 Linux 系统 (例如 RedHat 9 或 Fedora 4 等) 下进行实验, 那么通常我们只需要额外安装 Bochs 软件包即可。其他操作都可以利用 Linux 系统的普通命令来完成。

运行 Linux 0.1x 系统的最佳方法是使用 PC 仿真软件。目前市面上流行的 PC 仿真软件系统主要有 3 种: VMware 公司的 VMware Workstation 软件、Connectix 公司的 Virtual PC (现在已被微软收购) 和开放源代码软件 Bochs (发音与 'box' 相同)。这 3 种软件都虚拟或仿真了 Intel x86 硬件环境, 可以让我们在运行这些软件的系统平台上运行多种其他的“客户”操作系统。

就使用范围和运行性能来说, 这 3 种仿真软件还是具有一定的区别。Bochs 仿真了 x86 的硬件环境 (CPU 的指令) 及其外围设备, 因此很容易被移植到很多操作系统上或者不同体系结构的平台上。由于主要使用了仿真技术, 其运行性能和速度都要比其他两个软件要慢很多。Virtual PC 的性能则界于 Bochs 和 VMware Workstation 之间。它仿真了 x86 的大部分指令, 而其他部分则采用虚拟技术来实现。VMware Workstation 仅仿真了一些 I/O 功能, 而所有其他部分则是在 x86 实时硬件上直接执行。也就是说当客户操作系统在要求执行一条指令时, VMware 不是用仿真方法来模拟这条指令, 而是把这条指令“传递”给实际系统的硬件来完成。因此 VMware 是 3 种软件中运行速度和性能最高的一种。有关这 3 种软件之间的具体区别和性能差异, 请参考网上的评论文章 (http://www.osnews.com/story.php?news_id=1054)。

从应用方面来看, 如果仿真环境主要用于应用程序开发, 那么 VMware Workstation 和 Virtual PC 可能是比较好的选择。但是如果需要开发一些低层系统软件 (比如进行操作系统开发和调试、编译系统开发等), 那么 Bochs 就是一个很好的选择。使用 Bochs, 你可以知道被执行程序在仿真硬件环境中的具体状态和精确时序, 而非实际硬件系统执行的结果。这也是为什么很多操作系统开发者更倾向于使用 Bochs 的原因。因此本章主要介绍利用 Bochs 仿真环境运行 Linux 0.11 的方法。目前, Bochs 网站名是 <http://sourceforge.net/projects/bochs/>。你可以从上面下载到最新发布的 Bochs 软件系统以及很多已经制作好的可运行磁盘映像文件。

17.1 Bochs 仿真软件系统

Bochs 是一个能完全仿真 Intel x86 计算机的程序。它可以被配置成仿真 386、486、Pentium 或以上的新型 CPU。在执行的全过程, Bochs 仿真了所有的指令, 并且含有标准 PC 机外设所有的设备模块。由于 Bochs 仿真了整个 PC 环境, 因此在其中执行的软件会“认为”它是在一个真实的机器上运行。这种完全仿真的方法使得我们能在 Bochs 下不加修改地运行大量的软件系统。

Bochs 是 Kevin Lawton 于 1994 年开始采用 C++ 语言开发的软件系统, 被设计成能够在 Intel x86、PPC、Alpha、Sun 和 MIPS 硬件上运行。不管运行的主机采用的是何种硬件平台, Bochs 仍然能仿真 x86 的硬件平台。这种特性是其他两种仿真软件没有的。为了在被模拟的机器上执行任何活动, Bochs 需要与主机操作系统进行交互。当在 Bochs 显示窗口中按下一键时, 一个击键事件就会发送到键盘的设备处理模块中。当被模拟的机器需要从模拟的硬盘上执行读操作时, Bochs 就会从主机上的硬盘映像文件中执行读操作。

Bochs 软件的安装非常方便。你可以直接从 <http://bochs.sourceforge.net> 网站上下载到 Bochs 安装软件包。如果你所使用的计算机操作系统是 Windows, 则其安装过程与普通软件完全一样。安装好后会在 C 盘上生成一个目录: 'C:\Program Files\Bochs-2.2.1\' (其中版本号随不同的版本而不同)。如果你的系统是 RedHat 9 或其他 Linux 系统, 你可以下载 Bochs 的 RPM 软件包并按如下方法来安装:

```
user$ su
Password:
root# rpm -i bochs-2.2.1.i386.rpm
root# exit
user$ _
```

安装时需要有 root 权限, 否则你就得在自己的目录下重新编译 Bochs 系统。另外, Bochs 需要在 X11 环境下运行, 因此你的 Linux 系统中必须已经安装了 X Window 系统才能使用 Bochs。在安装好 Bochs 之后, 建议先使用 Bochs 中自带的 Linux dlx 系统程序包来测试和熟悉一下 Bochs 系统。另外, 也可以从 Bochs 网站上下载一些已经制作好的 Linux 磁盘映像文件。我们建议下载 Bochs 网站上的 SLS Linux 模拟系统: sls-0.99pl.tar.bz2 作为创建 Linux 0.1x 模拟系统的辅助平台。在我们制作新的硬盘映像文件时, 需要借助这些系统对硬盘映像文件进行分区和格式化操作。这个 SLS Linux 系统也可以直接从 www.oldlinux.org 网站下载: <http://oldlinux.org/Linux.old/bochs/sls-1.0.zip>。下载的文件解压后进入其目录并双击 bochsrc.bxrc 配置文件名¹²即可让 Bochs 运行 SLS Linux 系统。

有关重新编译 Bochs 系统或把 Bochs 安装到其他硬件平台上的操作方法, 请参考 Bochs 用户手册中的相关说明。

17.1.1 设置 Bochs 系统

为了在 Bochs 中运行一个操作系统, 最少需要以下一些资源或信息:

- bochs 执行文件;
- bios 映像文件 (通常称为 'BIOS-bochs-latest');
- vga bios 映像文件 (例如, 'VGABIOS-lgpl-latest');
- 至少一个引导启动磁盘映像文件 (软盘、硬盘或 CDROM 的映像文件)。

但是我们在使用过程中往往需要为运行系统预先设置一些参数。这些参数可以在命令行上传递给 Bochs 执行程序, 但通常我们都使用一个配置文件 (文件后缀为 .bxrc, 例如 Sample.bxrc) 为专门的一个应用来设置运行参数。下面说明 Bochs 配置文件的设置方法。

17.1.2 配置文件 *.bxrc

Bochs 使用配置文件中的信息来寻找所使用的磁盘映像文件、运行环境外围设备的配置以及其他一些模拟机器的设置信息。每个被仿真的系统都需要设置一个相应的配置文件。若所安装的 Bochs 系统是 2.1 或以后版本, 那么 Bochs 系统会自动识别后缀是 '.bxrc' 的配置文件, 并且在双击该文件图标时就会自动启动 Bochs 系统运行。例如, 我们可以把配置文件名取为 'bochsrc-0.11.bxrc'。在 Bochs 安装的主目录

¹² 如果配置文件名没有后缀 .bxrc, 请自行修改。例如原名为 bochsrc 修改成 bochsrc.bxrc。

下有一个名称为'bochsrc-sample.txt'的样板配置文件，其中列出了所有可用的参数设置，并带有详细的说明。下面简单介绍几个在实验中经常要修改的参数。

1. megs

用于设置被模拟系统所含内存容量。默认值是 32MB。例如，如果要把模拟机器设置为含有 128MB 的系统，则需要在配置文件中含有如下一行信息：

```
megs: 128
```

2. floppy (floppyb)

floppy 表示第一个软驱，**floppyb** 代表第二个软驱。如果需要一个软盘上来引导系统，那么 **floppy** 就需要指向一个可引导的磁盘。若想使用磁盘映像文件，那么我们就在该选项后面写上磁盘映像文件的名称。在许多操作系统中，Bochs 可以直接读写主机系统的软盘驱动器。若要访问这些实际驱动器中的磁盘，就使用设备名称（Linux 系统）或驱动器号（Windows 系统）。还可以使用 **status** 来表明磁盘的插入状态。**ejected** 表示未插入，**inserted** 表示磁盘已插入。下面是几个例子，其中所有盘均为已插入状态。若在配置文件中同时存在几行相同名称的参数，那么只有最后一行的参数起作用。

```
floppy: l_44=/dev/fd0, status=inserted      # Linux 系统下直接访问 1.44MB A 盘。
floppy: l_44=b:, status=inserted           # win32 系统下直接访问 1.44MB B 盘。
floppy: l_44=bootimage.img, status=inserted # 指向磁盘映像文件 bootimage.img。
floppyb: l_44=..\Linux\rootimage.img, status=inserted # 指向上级目录 Linux/下 rootimage.img。
```

3. ata0、ata1、ata2、ata3

这 4 个参数名用来启动模拟系统中最多 4 个 ATA 通道。对于每个启用的通道，必须指明两个 IO 基地址和一个中断请求号。默认情况下只有 **ata0** 是启用的，并且参数默认为下面所示的值：

```
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata2: enabled=1, ioaddr1=0x1e8, ioaddr2=0x3e0, irq=11
ata3: enabled=1, ioaddr1=0x168, ioaddr2=0x360, irq=9
```

4. ata0-master (ata0-slave)

ata0-master 用来指明模拟系统中第 1 个 ATA 通道(0 通道)上连接的第 1 个 ATA 设备(硬盘或 CDROM 等)；**ata0-slave** 指明第 1 个通道上连接的第 2 个 ATA 设备。例子如下所示，其中，设备配置的选项含义如表 17-1 所示。

```
ata0-master: type=disk, path=hd.img, mode=flat, cylinders=306, heads=4, spt=17, translation=none
ata1-master: type=disk, path=2G.cow, mode=vmware3, cylinders=5242, heads=16, spt=50, translation=echs
ata1-slave:  type=disk, path=3G.img, mode=sparse, cylinders=6541, heads=16, spt=63, translation=auto
ata2-master: type=disk, path=7G.img, mode=undoable, cylinders=14563, heads=16, spt=63, translation=lba
ata2-slave:  type=cdrom, path=iso.sample, status=inserted
ata0-master: type=disk, path="hdc-large.img", mode=flat, cylinders=487, heads=16, spt=63
ata0-slave:  type=disk, path="..\hdc-0.11.img", mode=flat, cylinders=121, heads=16, spt=63
```

表 17-1 设备配置的选项

选项	说明	可取的值
----	----	------

type	连接的设备类型	[disk cdrom]
path	映像文件路径名	
mode	映像文件类型, 仅对 disk 有效	[flat concat external dll sparse vmware3 undoable growing volatile]
cylinders	仅对 disk 有效	
heads	仅对 disk 有效	
spt	仅对 disk 有效	
status	仅对 cdrom 有效	[inserted ejected]
biosdetect	bios 检测类型	[none auto],仅对 ata0 上 disk 有效 [cmos]
translation	bios 进行变换的类型(int13),仅对 disk 有效	[none lba large rechs auto]
model	确认设备 ATA 命令返回的字符串	

在配置 ATA 设备时, 必须指明连接设备的类型 **type**, 可以是 **disk** 或 **cdrom**。还必须指明设备的“路径名” **path**。“路径名”可以是一个硬盘映像文件、CDROM 的 iso 文件或者直接指向系统的 CDROM 驱动器。在 Linux 系统中, 可以使用系统设备作为 Bochs 的硬盘, 但由于安全原因, 在 windows 下不赞成直接使用系统上的物理硬盘。

对于类型是 **disk** 的设备, 选项 **path**、**cylinders**、**heads** 和 **spt** 是必须的。对于类型是 **cdrom** 的设备, 选项 **path** 是必须的。

磁盘变换方案(在传统 int13 bios 功能中实现, 并且用于象 DOS 这样的老式操作系统)可以定义为:

- ◆ **none**: 无需变换, 适用于容量小于 528MB (1032192 个扇区) 的硬盘;
- ◆ **large**: 标准比特移位算法, 用于容量小于 4.2GB (8257536 个扇区) 的硬盘;
- ◆ **rechs**: 修正移位算法, 使用 15 磁头的伪物理硬盘参数, 适用于容量小于 7.9GB (15482880 个扇区) 的硬盘;
- ◆ **lba**: 标准 lba-辅助算法。适用于容量小于 8.4GB (16450560 个扇区) 的硬盘;
- ◆ **auto**: 自动选择最佳变换方案。(如果模拟系统启动不了就应该改变)。

mode 选项用于说明如何使用硬盘映像文件。它可以是以下模式之一:

- ◆ **flat**: 一个平坦顺序文件;
- ◆ **concat**: 多个文件;
- ◆ **external**: 由开发者专用, 通过 C++类来指定;
- ◆ **dll**: 开发者专用, 通过 DLL 来使用;
- ◆ **sparse**: 可堆砌的、可确认的、可退回的;
- ◆ **vmware3**: 支持 vmware3 的硬盘格式;
- ◆ **undoable**: 具有确认重做的平坦文件;
- ◆ **growing**: 容量可扩展的映像文件;
- ◆ **volatile**: 具有易变重做的平坦文件。

以上选项的默认值是:

mode=flat, biosdetect=auto, translation=auto, model="Generic 1234"

5. boot

boot 用来定义模拟机器中用于引导启动的驱动器。可以指定是软盘、硬盘或者 CDROM。也可以使用驱动器号 'c' 和 'a'。例子如下:

```
boot: a
```

```
boot: c
boot: floppy
boot: disk
boot: cdrom
```

6. ips

ips (Instructions Per Second) 指定每秒钟仿真的指令条数。这是 Bochs 在主机系统中运行的 IPS 数值。这个值会影响模拟系统中与时间有关的很多事件。例如改变 IPS 值会影响到 VGA 更新的速率以及其他一些模拟系统评估值。因此需要根据所使用的主机性能来设定该值。可参考表 17-2 进行设置。

表 17-2 每秒种仿真指令数

速度	机器配置	IPS 典型值
650Mhz	Athlon K-7 with Linux 2.4.x	2 to 2.5 million
400Mhz	Pentium II with Linux 2.0.x	1 to 1.8 million
166Mhz	64bit Sparc with Solaris 2.x	0.75 million
200Mhz	Pentium with Linux 2.x	0.5 million

例如：

```
ips: 1000000
```

7. log

指定 log 的路径名可以让 Bochs 记录执行的一些日志信息。如果在 Bochs 中运行的系统发生不能正常运行的情况就可以参考其中的信息来找出基本原由。log 通常设置为：

```
log: bochsout.txt
```

17.2 在 Bochs 中运行 Linux 0.1x 系统

若要运行一个 Linux 类操作系统，那么除了需要内核代码以外，我们还需要一个根文件系统。根文件系统通常是一个存放 Linux 系统运行时必要文件（例如系统配置文件和设备文件等）和存储数据文件的外部设备。在现代 Linux 操作系统中，内核代码映像文件（bootimage）保存在根文件系统（root fs）中。系统引导启动程序会从这个根文件系统设备上把内核执行代码加载到内存中去运行。

不过内核映像文件和根文件系统并不要求一定要存放在同一个设备上，即无须存放在一个软盘或硬盘的同一个分区中。对于只使用软盘的情况，由于软盘容量方面的限制，通常就把内核映像文件与根文件系统分别单独放在一个盘片中，存放可引导启动的内核映像文件的软盘被称为内核引导启动盘文件（bootimage）；存放根文件系统的软盘就被称作根文件系统映像文件（rootimage）。当然我们也可以从软盘中加载内核映像文件而使用硬盘中的根文件系统，或者让系统直接从硬盘开始引导启动系统，即从硬盘的根文件系统中加载内核映像文件并使用硬盘中的根文件系统。

本节主要介绍如何在 Bochs 中运行几种已经设置好的 Linux 0.1x 系统，并且说明相关配置文件中几个主要参数的设置。首先我们从网站上下载一个如下 Linux 0.1x 系统软件包到桌面上：

<http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-050518.zip>

软件包名称中的最后 6 位数字是日期信息。通常应该选择下载日期最新的一个软件包。下载完毕后可以使用如 `unzip`、`winzip` 或 `rar` 等一般通用解压缩程序来解开。注意，你需要大约 150MB 的磁盘空间来解开这个压缩文件。

17.2.1 软件包中文件说明

解开 `linux-0.11-devel-050518.zip` 这个文件后会生成一个名称为 `linux-0.11-devel-050518` 的目录。进入该目录后我们可以看到其中大约有如下 20 个文件。

```
[root@plinux]# ls -o -g -l
total 136348
-rw-rw-rw- 1 2441509 May 18 14:20 bochs-2.2.1.i586.rpm
-rw-r--r-- 1 3198301 May 18 14:17 Bochs-2.2.pre4.exe
-rw-rw-rw- 1 14187 May 18 14:24 bochsout.txt
-rw-rw-rw- 1 18165 Sep 23 2004 bochsrc-fda.bxrc
-rw-rw-rw- 1 18125 Sep 23 2004 bochsrc-fdb.bxrc
-rw-rw-rw- 1 18050 May 18 14:24 bochsrc-hdboot.bxrc
-rw-rw-rw- 1 18050 May 18 14:24 bochsrc-hd.bxrc
-rw-rw-rw- 1 121344 Aug 17 2004 bootimage-0.11
-rw-rw-rw- 1 121344 Mar 4 2004 bootimage-0.11-fd
-rw-rw-rw- 1 121856 Apr 29 2004 bootimage-0.11-hd
-rw-rw-rw- 1 145920 Aug 29 2002 bootimage-0.12-fd
-rw-rw-rw- 1 150528 Feb 23 2004 bootimage-0.12-hd
-rw-r--r-- 1 65 Jul 26 2004 debug.bat
-rw-rw-rw- 1 1474560 Oct 5 2004 disk.a.img
-rw-rw-rw- 1 1474560 Apr 29 2004 disk.b.img
-rw-rw-rw- 1 934577 Mar 29 2004 gcc-lib-1.40.taz
-rw-rw-rw- 1 127631360 May 18 14:24 hdc-0.11-new.img
-rw-rw-rw- 1 5901 May 18 14:23 README
-rw-rw-rw- 1 1474560 May 18 13:52 rootimage-0.11
-rw-rw-rw- 1 17771 Oct 5 2004 SYSTEM.MAP
[root@plinux]#
```

这个软件包中包含有 2 个 Bochs 安装程序、4 个不同的 Bochs 配置文件、5 个内核映像（Image）文件；一个软盘和一个硬盘根文件系统映像文件以及其他一些有用文件。其中的 `README` 文件简要说明了各个文件的用途。这里我们再稍微详细说明一下各个文件的用途。

- `bochs-2.2.1.i586.rpm` 是 Linux 操作系统下的 Bochs 安装程序。
- `Bochs-2.2.pre4.exe` 是 windows 操作系统平台下的 Bochs 安装程序。在运行 Linux 0.11 系统之前我们需要首先在机器上安装 Bochs 系统。最新版的 Bochs 软件可以网站：<http://sourceforge.net/projects/bochs/> 上下载。
- `bochsout.txt` 是 Bochs 系统运行时自动产生的日志文件。其中包含有 Bochs 运行时各种状态信息。在运行 Bochs 遇到问题时，可以查看这个文件的内容来初步断定问题的原因。
- `bochsrc-fda.bxrc` 是 Bochs 的配置文件。这个配置文件用于从 Bochs 虚拟 A 盘（`/dev/fd0`）启动 Linux 0.11 系统，即内核映像文件已设置在虚拟 A 盘中，并且要求随后根文件系统被替换插入当前虚拟启动驱动器中。在引导启动过程中，它会要求我们在 A 盘中“插入”软盘根文件系统盘（`rootimage-0.11`）。这个配置文件将使用内核映像文件 `bootimage-0.11`。双击这个配置文件即可运行该配置的 Linux 0.11 系统。
- `bochsrc-fdb.bxrc` 也是运行 Bochs 的配置文件。该配置文件已经把软盘根文件系统盘（`rootimage-0.11`）设置成在 B 盘（`/dev/fd1`）中，因此当显示要求我们插入根文件系统盘时可直接

按回车。这个配置文件将使用内核映像文件 `bootimage-0.11-fd`。双击这个配置文件即可运行该配置的 Linux 0.11 系统。

- `bochsrc-hd.bxrc` 这也是一个设置成从 A 盘启动的配置文件，但是会使用硬盘（C 盘）映像文件（`hdc-0.11-new.img`）中的根文件系统。这个配置文件将使用内核映像文件 `bootimage-0.11-hd`。双击这个配置文件即可运行该配置的 Linux 0.11 系统。
- `bochsrc-hdboot.bxrc` 这是一个从虚拟硬盘（`hdc-0.11-new.img`）引导启动 Linux 0.11 系统的配置文件，因此所使用的引导启动内核映像文件已在虚拟硬盘中（`/usr/src/linux/Image`）。
- `bootimage-0.11` 是编译内核生成的映像（Image）文件。其中包含了整个内核的代码和数据，包括软盘启动引导扇区的代码。双击这个配置文件即可运行该配置的 Linux 0.11 系统。
- `bootimage-0.11-fd` 也是编译内核生成的映像（Image）文件。与 `bootimage-0.11` 文件的主要区别在于其中引导扇区（最初 512 字节）中第 509、510 字节的根文件系统设备号已被设置成 B 盘（`/dev/fd1`），设备号是 `0x021D`。其他方面完全和 `bootimage-0.11` 文件一样。
- `bootimage-0.11-hd` 是用于使用虚拟硬盘上根文件系统的内核映像文件，即该文件的第 509、510 字节的根文件系统设备号已被设置成 C 盘第 1 个分区（`/dev/hd1`），设备号是 `0x0301`。
- `bootimage-0.12-fd` 是 Linux 0.12 内核的映像文件，其作用同 `bootimage-0.11`。
- `bootimage-0.12-hd` 也是 Linux 0.12 内核的映像文件，其作用同 `bootimage-0.11-hd`。
- `debug.bat` 是 windows 平台上启动 Bochs 调试功能的批处理程序。请注意，你可能需要根据 Bochs 安装的具体目录来修改其中的路径名。另外，默认情况下在 Linux 系统上安装运行的 Bochs 系统不包含调试功能。你可以直接使用 Linux 系统中的 `gdb` 程序进行调试。若还是想利用 Bochs 的调试功能，那么你就需要下载 Bochs 的源代码自己进行定制编译。
- `diska.img` 和 `diskb.img` 是两个 DOS 格式的软盘映像文件。其中包含了一些工具程序。在 Linux 0.11 中可以使用 `mcoppy` 等命令来访问这两个文件。当然在访问之前需要动态“插入”相应的盘片。在双击 `bochsrc-fda.bxrc`、或 `bochsrc-hd.bxrc` 或 `bochsrc-hdboot.bxrc` 配置文件设置的 Linux 0.11 系统时，B 盘中已经“插入”了 `diskb.img` 盘。
- `gcclib-1.4.0.taz` 是 Linux 0.11 系统中使用了 GNU gcc 1.40 编译系统。放在这里只是为了让大家练习从硬盘映像文件中导入/导出的使用方法。因为硬盘映像文件中已经安装了这个 gcc 开发环境。
- `hdc-0.11-new.img` 就是上面提到的虚拟硬盘映像文件。其中第 1 个分区中是一个 MINIX 文件系统 1.0 类型的根文件系统，第 2 个分区也是一个 MINIX 1.0 文件系统，但没有存放任何文件。你可以使用 `mount` 命令加载并使用这个额外的空间。
- `rootimage-0.11` 是软盘上的根文件系统盘。当使用 `bochsrc-fda.bxrc` 或 `bochsrc-fdb.bxrc` 来运行 Linux 0.11 系统时，就会用到这个根文件系统盘。
- `SYSTEM.MAP` 文件是编译 Linux 0.11 内核时生成的内核内存存储位置信息文件。在调试内核时，该文件的内容非常有用。

17.2.2 安装 Bochs 模拟系统

软件包中的 `bochs-2.2.1.i586.rpm` 文件是 RedHat Linux 系统下的 Bochs 安装程序，`Bochs-2.2.pre4.exe` 是 windows 操作系统上的 Bochs 安装程序。最新版的 Bochs 软件总是可以在下面网站位置下载：

<http://sourceforge.net/projects/bochs/>

若我们是在 Linux 系统中，那么就可在命令行上运行 `rpm` 命令或者在 X window 中直接双击第 1 个文件来安装 Bochs：

```
rpm -i bochs-2.2.pre4-1.i586.rpm
```

若是在 Windows 系统下，那么直接双击 **Bochs-2.2.pre4.exe** 文件名就可来安装 Bochs 系统。在安装完后请根据安装的具体目录修改用于调试内核的批处理文件 **debug.bat**。另外，在下面的实验过程和例子中，我们主要以 Windows 平台进行介绍 Bochs 的使用方法。

17.2.3 运行 Linux 0.1x 系统

在 Bochs 中运行 Linux 0.1x 系统非常简单，你只要双击相应的 Bochs 配置文件 (*.bxrc) 即可开始运行。每个配置文件中已经设置好了运行时模拟的 PC 机环境。你可以利用任何文本编辑器（例如 **notepad.exe**）来编辑配置文件。对于运行 Linux 0.11 系统，相应的配置文件中通常只需要包含以下几行必要信息：

```
romimage: file=$BXSHARE\BIOS-bochs-latest, address=0xf0000
vgaromimage: $BXSHARE\VGABIOS-elpin-2.40
megs: 16
floppya: 1_44="bootimage-0.11", status=inserted
ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63
boot: a
```

前两行指明所模拟的 PC 机的 ROM BIOS 和 VGA 显示 ROM 程序，一般用不着修改。第 3 行指明 PC 机的物理内存容量，这里设置为 16MB。因为默认的 Linux 0.11 内核最多只支持 16MB 内存，所以设置大了也不起作用。参数 **floppya** 指定模拟 PC 机的软盘驱动器 A 使用 1.44MB 盘类型，并且这里已经设置成使用 **bootimage-0.11** 软盘映像文件，并且是在插入状态。对应的 **floppyb** 用来指明 B 盘中使用或插入的软盘映像文件。参数 **ata0-master** 用于指定模拟 PC 机上挂接的虚拟硬盘容量和硬盘参数。这些硬盘参数的具体含义请参见前面的描述。另外还有 **ata0-slave** 用来指定第 2 块虚拟硬盘使用的映像文件和参数。最后的 **boot** 用来指定启动的驱动器。可以设置成从 A 盘或从 C 盘（硬盘）启动。这里设置成从 A 盘启动(a)。

1. 使用 bochsrc-fda.bxrc 配置文件运行 Linux 0.11 系统。

即从软盘启动 Linux 0.11 系统并且在当前驱动器中使用根文件系统。这种运行 Linux 0.11 系统的方式仅使用两个软盘：**bootimage-0.11** 和 **rootimage-0.11**。上面列出的几行配置文件内容就是 **bochsrc-fda.bxrc** 中的基本设置。当双击这个配置文件运行 Linux 0.11 系统时，Bochs 显示主窗口中会出现提示信息，见图 17-1 所示。由于 **bochsrc-fda.bxrc** 把 Linux 0.11 的运行环境配置成从 A 盘启动，并且所设置使用的内核映像文件 **bootimage-0.11** 会要求根文件系统在当前用于启动的驱动器（A 盘）中，所以内核会显示一条要求我们“取出”内核启动映像文件 **bootimage-0.11** 并“插入”根文件系统的信息。此时我们可以利用窗口上左上方的 A 盘图标来“更换”A 盘。单击这个图标，并把其中原映像文件名（**bootimage-0.11**）修改成 **rootimage-0.11**，我们就完成了软盘更换操作。此后单击“OK”按钮关闭该对话框后，再按回车键就可以让内核加载软盘上根文件系统，最后出现命令提示行，见图 17-2 所示。

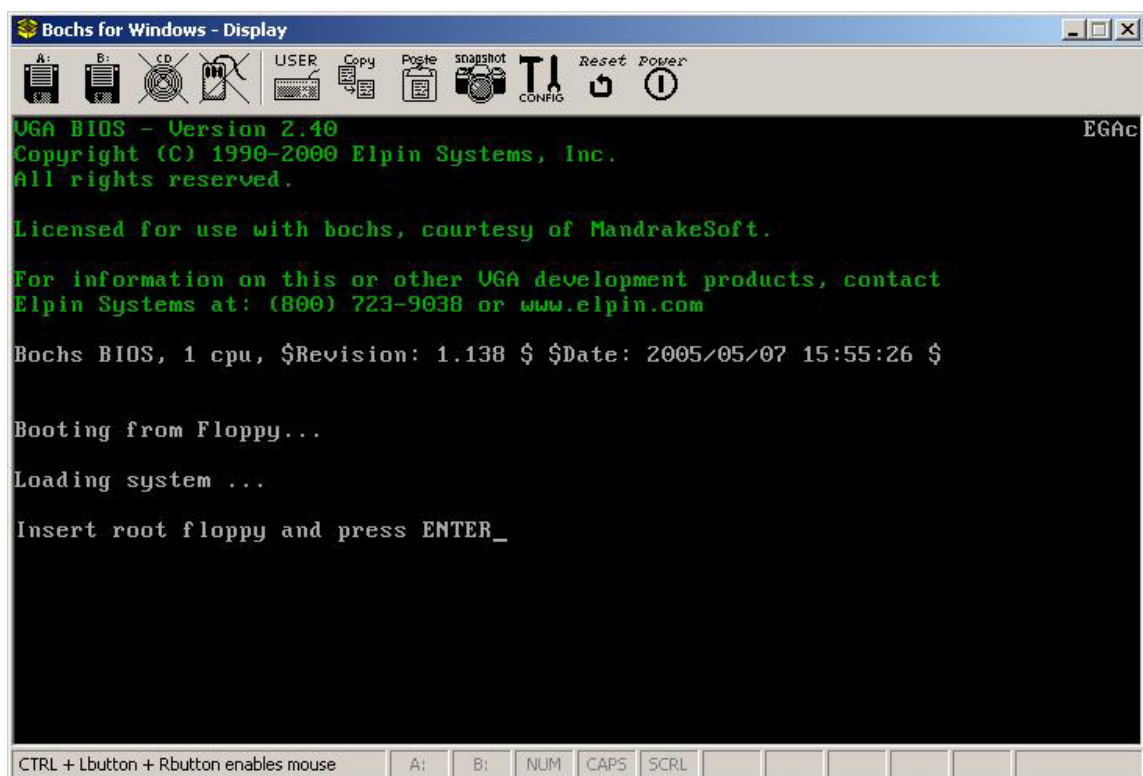


图 17-1 从软盘引导启动并运行在软盘中的根文件系统

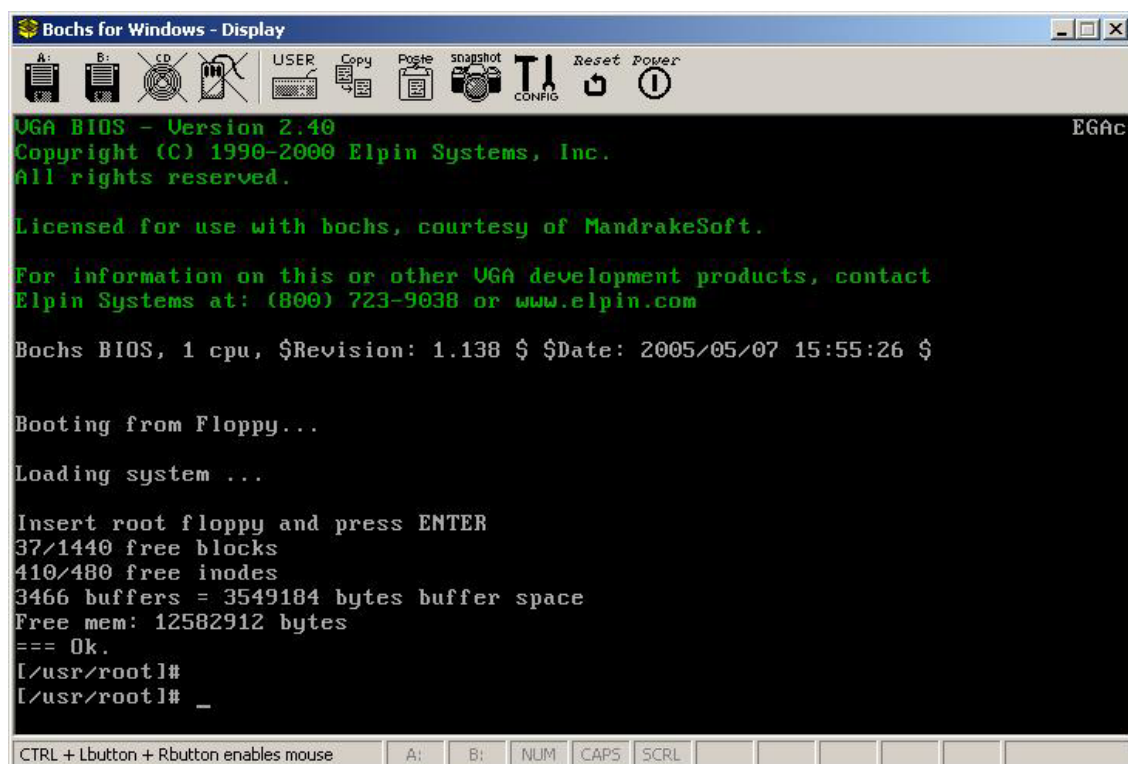


图 17-2 “更换”软盘并按回车键继续运行

2. 使用 bochsrc-fdb.bxrc 配置文件运行 Linux 0.11 系统

这个配置文件已经把虚拟 A 盘设置成“插入”了 `bootimage-0.11-fd` 内核映像文件，而虚拟 B 盘中已经“插入”了根文件系统 `rootimage-0.11` 映像文件。而且在 `bootimage-0.11-fd` 文件中前 512 字节的磁盘引导扇区中的根文件系统设备号已经设置成 B 盘，即 `bootimage-0.11-fd` 文件的第 509、510 字节被设置成 B 盘的设备号 `0x021D`（即 `0x1D,0x02`），此时内核运行时会自动从 B 盘上加载根文件系统。因此双击 `bochsrc-fdb.bxrc` 文件名就可以直接运行 Linux 0.11 系统，并得到图 17-2 的画面。

3. 使用 `bochsrc-hd.bxrc` 配置文件运行 Linux 0.11 系统

该配置文件会从启动软盘（A 盘）中加载 Linux 0.11 的内核映像文件 `bootimage-0.11-hd`，并且使用硬盘映像文件 `hdc-0.11-new.img` 第 1 个分区中的根文件系统。因为 `bootimage-0.11-hd` 文件中的第 509、510 字节已经被设置成 C 盘第 1 个分区的设备号 `0x0301`（即 `0x01,0x03`），因此内核初始化运行时会自动从 C 盘第 1 个分区中加载根文件系统。双击 `bochsrc-hd.bxrc` 文件名可以直接运行 Linux 0.11 系统，并同样会直接得到图 17-2 的画面。

4. 使用 `bochsrc-hdboot.bxrc` 配置文件运行 Linux 0.11 系统

使用 `bochsrc-hdboot.bxrc` 配置文件可以让 Linux 0.11 系统象现代 Linux 系统使用 LiLo 或 Grub 引导程序直接从硬盘启动系统一样来运行，但 Linux 0.11 使用的是原来 MINIX 操作系统的引导程序 `shoelace`。该引导程序会直接从虚拟硬盘的引导扇区代码开始执行，并把存放在虚拟硬盘根文件系统中的内核映像文件（默认为 `/usr/src/linux/Image`）加载到内存中去执行，同时就使用硬盘上的这个根文件系统。

双击 `bochsrc-hdboot.bxrc` 文件名可以直接运行 Linux 0.11 系统，但出现的画面与前面几幅略有不同。首先它会显示图 17-3 的画面。其中列出了虚拟硬盘上 4 个分区的基本参数。左面具有星号 '*' 的分区是默认活动分区。如果不按任何键而稍等几秒钟，Bochs 模拟的 PC 虚拟机就会尝试从这个分区引导系统。如果其他分区上也有可启动的操作系统，那么我们可以在这段延迟时间内键入分区号直接从指定的分区引导启动指定的系统。在我们现在的情况下，只有第 1 个分区存放着可启动的 Linux 0.11 系统，因此当按下数字 1 或者不按任何键而稍等几秒中后，Bochs 就会运行 Linux 0.11 系统，见图 17-4 所示。按任何其他数字键不会有任何反应。

此时 Bochs 中运行的内核映像文件是根文件系统中的 `/usr/src/linux/Image` 文件。你也可以通过修改 `shoelace` 引导程序的配置文件 `/etc/config` 重新设置用于引导启动的内核映像文件。

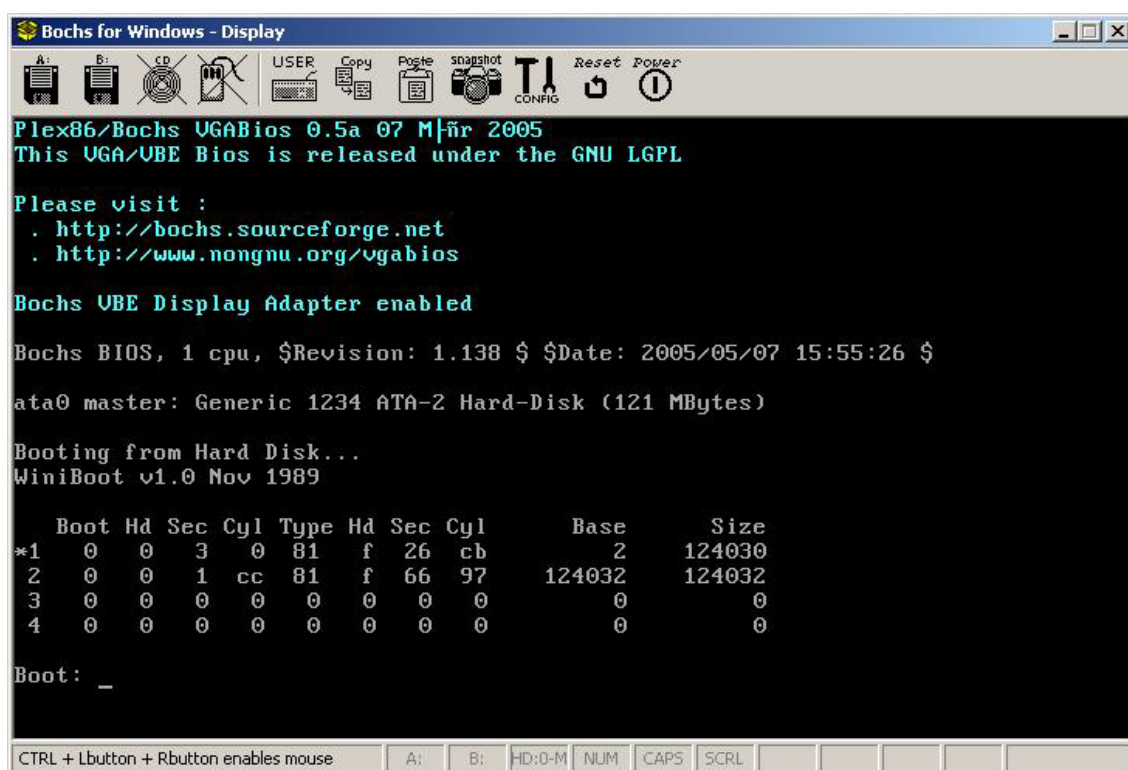


图 17-3 使用 shoelace 引导程序直接从硬盘启动

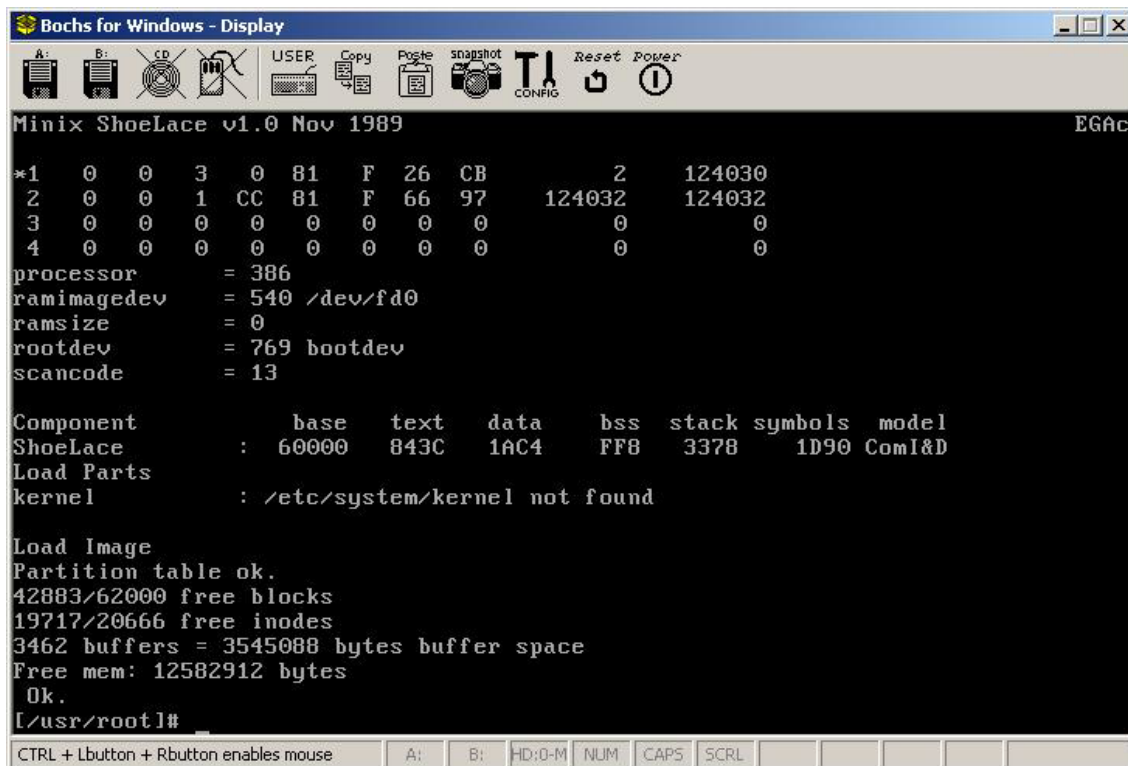


图 17-4 使用 shoelace 引导启动完成后

17.3 访问磁盘映像文件中的信息

Bochs 使用磁盘映像文件来仿真被模拟系统中外部存储设备，被模拟操作系统中的所有文件均以软盘或硬盘设备中的格式保存在映像文件中。由此就带来了主机操作系统与 Bochs 中被模拟系统之间交换信息的问题。虽然 Bochs 系统能被配置成直接使用主机的软盘驱动器、CDROM 驱动器等物理设备来运行，但是利用这种信息交换方法比较烦琐。因此最好能够直接读写 Image 文件中的信息。如果需要往被模拟的操作系统中添加文件，就把文件存入 Image 文件中。如果要取出其中的文件就从 Image 文件中读出。但由于保存在 Image 文件中的信息不仅是按照相应的软盘或硬盘格式存放，而且还以一定的文件系统格式存放。因此访问 Image 文件中信息的程序必须能够识别其中的文件系统才能操作。对于本章应用来说，我们需要一些工具来识别 Image 文件中的 MINIX 和（或）DOS 文件系统格式。

总体来说，如果与模拟系统交换的文件长度比较小，我们可以采用软盘 Image 文件作为交换媒介。如果有大批量文件需要从模拟系统中取出或放入模拟系统，那么我们可以利用现有 Linux 系统来操作。下面就从这两个方面讨论可采用的几种方法。

利用磁盘映像读写工具访问软盘映像文件中的信息（小文件或分割的文件）

在 Linux 主环境中利用 loop 设备访问硬盘映像文件中的信息。（大批量信息交换）

利用 iso 格式文件进行信息交换（大批量信息交换）

17.3.1 使用 WinImage 工具软件

使用软盘 Image 文件，我们可以与模拟系统进行少量文件的交换。前提条件是被模拟系统支持对 DOS 格式软盘进行读写，例如通过使用 mtools 软件。mtools 是 UNIX 类系统中读写访问 MSDOS 文件系统中文件的程序。该软件模拟或仿真了常用的 MSDOS 命令，如 copy、dir、cd、format、del、md 和 rd 等。在这些名称前加上字母 m 就是 mtools 中的对应命令。下面以实例来说明具体的操作方法。

在读写文件之前，首先需要根据前面描述的方法准备一个 1.44MB Image 文件（文件名假设是 diskb.img）。并修改 Linux 0.11 的 bochs.bxrc 配置文件，在 floppyb 参数下增加以下一行信息：

```
floppyb: 1_44="diskb.img", status=inserted
```

也即给模拟系统增加第 2 个 1.44MB 软盘设备，并且该设备对应的 Image 文件名是 diskb.img。

如果想把 Linux 0.11 系统中的某个文件取出来，那么现在可以双击配置文件图标开始运行 Linux 0.11 系统。在进入 Linux 0.11 系统后，使用 DOS 软盘读写工具 mtools 把 hello.c 文件写到第 2 个软盘 Image 中。如果软盘 Image 是使用 Bochs 创建的或还没有格式化过，可以使用 mformat b:命令首先进行格式化。

```
[/usr/root]# mcopy hello.c b:
Copying HELLO.C
[/usr/root]# mdir b:
Volume in drive B has no label
Directory for B:/

HELLO    C           74    4-30-104   4:47p
      1 File(s)    1457152 bytes free
[/usr/root]# _
```

现在退出 Bochs 系统，并使用 WinImage 打开 diskb.img 文件，在 WinImage 的主窗口中会有一个 hello.c 文件存在。用鼠标选中该文件并拖到桌面上即完成了取文件的整个操作过程。如果需要把某个文件

输入到模拟系统中，那么操作步骤正好与上述相反。另外请注意，WinImage 只能访问和操作具有 DOS 格式的盘片文件，它不能访问其他 MINIX 文件系统等格式的盘片文件。

17.3.2 利用现有 Linux 系统

现有 Linux 系统（例如 RedHat 9）能够访问多种文件系统，包括利用 loop 设备访问存储在文件中的文件系统。对于软盘 Image 文件，我们可以直接使用 mount 命令来加载 Image 中的文件系统进行读写访问。例如我们需要访问 rootimage-0.11 中的文件，那么只要执行以下命令。

```
[root@plinux images]# mount -t minix rootimage-0.11 /mnt -o loop
[root@plinux images]# cd /mnt
[root@plinux mnt]# ls
bin dev etc root tmp usr
[root@plinux mnt]# _
```

其中 mount 命令的 -t minix 选项指明所读文件系统类型是 MINIX，-o loop 选项说明通过 loop 设备来加载文件系统。若需要访问 DOS 格式软盘 Image 文件，只需把 mount 命令中的文件类型选项 minix 换成 msdos 即可。

如果想访问硬盘 Image 文件，那么操作过程与上述不同。由于软盘 Image 文件一般包含一个完整文件系统的映像，因此可以直接使用 mount 命令加载软盘 Image 中的文件系统，但是硬盘 Image 文件中通常含有分区信息，并且文件系统是在各个分区中建立的。也即我们可以把硬盘中的每个分区看成是一个完整的“大”软盘。

因此，为了访问一个硬盘 Image 文件某个分区中的信息，我们需要首先了解这个硬盘 Image 文件中分区信息，以确定需要访问的分区在 Image 文件中的起始偏移位置。关于硬盘 Image 文件中的分区信息，我们可以在模拟系统中使用 fdisk 命令查看，也可以利用这里介绍的方法查看。这里以下面软件包中包括的硬盘 Image 文件 hdc-0.11.img 为例来说明访问其中第 1 个分区中文件系统的方法。

<http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040329.zip>

这里需要用到 loop 设备设置与控制命令 losetup。该命令主要用于把一个普通文件或一个块设备与 loop 设备相关联，或用于释放一个 loop 设备、查询一个 loop 设备的状态。该命令的详细说明请参照在线手册页。

首先执行下面命令把 hdc-0.11.img 文件与 loop1 相关联，并利用 fdisk 命令查看其中的分区信息。

```
[root@plinux devel]# losetup /dev/loop1 hdc-0.11.img
[root@plinux devel]# fdisk /dev/loop1
Command (m for help): x                                # 进入扩展功能菜单
Expert command (m for help): p                            # 显示分区表
Disk /dev/loop1: 16 heads, 63 sectors, 121 cylinders

Nr AF  Hd Sec  Cyl  Hd Sec  Cyl    Start    Size ID
 1 80   1  1    0  15  63   119        1 120959 81
 2 00   0  0    0   0  0    0    0         0 00
 3 00   0  0    0   0  0    0    0         0 00
 4 00   0  0    0   0  0    0    0         0 00
Expert command (m for help): q
[root@plinux devel]# _
```

从上面 `fdisk` 给出的分区信息可以看出, 该 `Image` 文件仅含有 1 个分区。记下该分区的起始扇区号(也即分区表中 `Start` 一栏的内容)。如果你需要访问具有多个分区的硬盘 `Image`, 那么你就需要记住相关分区的起始扇区号。

接下来, 我们先使用 `losetup` 的 `-d` 选项把 `hdc-0.11.img` 文件与 `loop1` 的关联解除, 重新把它关联到 `hdc-0.11.img` 文件第 1 个分区的起始位置处。这需要使用 `losetup` 的 `-o` 选项, 该选项指明关联的起始字节偏移位置。由上面分区信息可知, 这里第 1 个分区的起始偏移位置是 $1 * 512$ 字节。在把第 1 个分区与 `loop1` 重新关联后, 我们就可以使用 `mount` 命令来访问其中的文件了。

```
[root@plinux devel]# losetup -d /dev/loop1
[root@plinux devel]# losetup -o 512 /dev/loop1 hdc-0.11.img
[root@plinux devel]# mount -t minix /dev/loop1 /mnt
[root@plinux devel]# cd /mnt
[root@plinux mnt]# ls
bin dev etc image mnt tmp usr var
[root@plinux mnt]# _
```

在对分区中文件系统访问结束后, 最后请卸载和解除关联。

```
[root@plinux mnt]# cd
[root@plinux root]# umount /dev/loop1
[root@plinux root]# losetup -d /dev/loop1
[root@plinux root]# _
```

17.4 编译运行简单内核示例程序

前面 80386 保护模式及其编程一章中给出了一个简单多任务内核示例程序, 我们称之为 `Linux 0.00` 系统。它含有两个运行在特权级 3 上的任务, 分别会在屏幕上循环显示字符 `A` 和 `B`, 并且在时钟定时控制下执行任务切换操作。在本书网站上给出了已经配置好的能在 `Bochs` 模拟环境下运行的软件包:

<http://oldlinux.org/Linux.old/bochs/linux-0.00-050613.zip>

<http://oldlinux.org/Linux.old/bochs/linux-0.00-041217.zip>

我们可以下载以上任何一个来进行实验。其中第 1 个软件包中给出的程序与这里描述的相同, 第 2 个软件包中的程序稍有不同(内核 `head` 代码直接在 `0x10000` 处运行), 但是原理完全一样。这里我们将以第 1 个软件包中的程序为例进行说明。第 2 个软件包请读者自己进行实验分析。

使用解压缩软件解开 `linux-0.00-050613.zip` 软件包后, 会在当前目录中生成一个 `linux-0.00` 子目录。我们可以看到这个软件包包含有以下几个文件:

1. `linux-0.00.tar.gz` - 源程序压缩文件;
2. `linux-0.00-rh9.tar.gz` - 源程序压缩文件;
3. `Image` - 内核引导启动映像文件;
4. `bochsrc-0.00.bxrc` - `Bochs` 配置文件;
5. `rawrite.exe` - `Windows` 下把 `Image` 写入软盘的程序。
6. `README` - 软件包说明文件;

第 1 文件 `linux-0.00.tar.gz` 是内核示例源程序的压缩文件，可以在 Linux 0.1x 系统中编译产生内核 Image 文件。第 2 个也是内核示例源程序的压缩文件，但其中的源程序可在 RedHat 9 Linux 系统下进行编译。第 3 个文件 `Image` 是源程序编译得到的可运行代码的 1.44MB 软盘映像文件。第 4 个文件 `bochsrc-0.00.bxrc` 是 Bochs 环境下运行时使用的 Bochs 配置文件，有关虚拟 PC 机模拟软件 Bochs 的安装和使用，请参考最后一章中的内容。如果你的系统上已经安装了虚拟 PC 机模拟软件 Bochs，那么只要用鼠标双击 `bochsrc-0.00.bxrc` 文件名就可以运行 `Image` 中的内核代码。第 5 个是 DOS 或 Windows 系统中把软盘映像文件写入软盘用的工具程序。我们可以直接运行 `RAWRITE.EXE` 程序并根据提示把这里的内核映像文件 `Image` 写入一张 1.44MB 软盘中来运行。

上面给出的内核示例的源程序就包括在 `linux-0.00-tar.gz` 文件中。解压这个文件会生成一个包含源程序文件的子目录，其中除了 `boot.s` 和 `head.s` 程序以外，还包含一个 `Makefile` 文件。由于 `as86/ld86` 编译链接产生的 `boot` 文件开始部分含有 32 字节的 MINIX 执行文件头部信息，而 `as/ld` 编译连接出的 `head` 文件开始部分包括 1024 字节的 `a.out` 格式头部信息，因此在生成内核 `Image` 文件时我们利用两条 `dd` 命令分别去掉两者头部信息并把它们合成内核映像 `Image` 文件。

在源代码目录中直接执行 `make` 命令即会生成 `Image` 文件。如果已经执行过 `make` 命令，那么请先执行 '`make clean`'，然后再执行 `make` 命令。

```
[/usr/root/linux-0.0]# ls -l
total 9
-rw----- 1 root    root          487 Jun 12 19:25 Makefile
-rw----- 1 root    4096        1557 Jun 12 18:55 boot.s
-rw----- 1 root    root        5243 Jun 12 19:01 head.s
[/usr/root/linux-0.0]# make
as86 -O -a -o boot.o boot.s
ld86 -O -s -o boot boot.o
gas -o head.o head.s
gld -s -x -M head.o -o system > System.map
dd bs=32 if=boot of=Image skip=1
16+0 records in
16+0 records out
dd bs=512 if=system of=Image skip=2 seek=1
16+0 records in
16+0 records out
[/usr/root/linux-0.0]#
```

若要把 `Image` 复制到 A 盘映像文件中或者一个真实的软盘中，那么我们可以再象下面一样执行命令 '`make disk`'。不过在执行该命令之前，若是在 Bochs 下 Linux 0.11 系统中执行的编译过程，那么请先复制保存你的启动映像盘文件（例如 `bootimage-0.11-hd`），以便测试完后恢复 Linux 0.11 系统的启动映像文件。

```
[/usr/root/linux-0.0]# ls
Image      System.map  boot.o      head.o      system
Makefile   boot        boot.s      head.s
[/usr/root/linux-0.0]# make disk
dd bs=8192 if=Image of=/dev/fd0
1+1 records in
1+1 records out
sync;sync;sync
[/usr/root/linux-0.0]#
```

若要运行这个内核示例，我们可以用鼠标直接单击 Bochs 窗口上的 RESET 图标。其运行情况见图 4-43 所示。此后若要恢复运行 Linux 0.11 系统，那么请用刚才复制保存的映像文件覆盖启动文件。

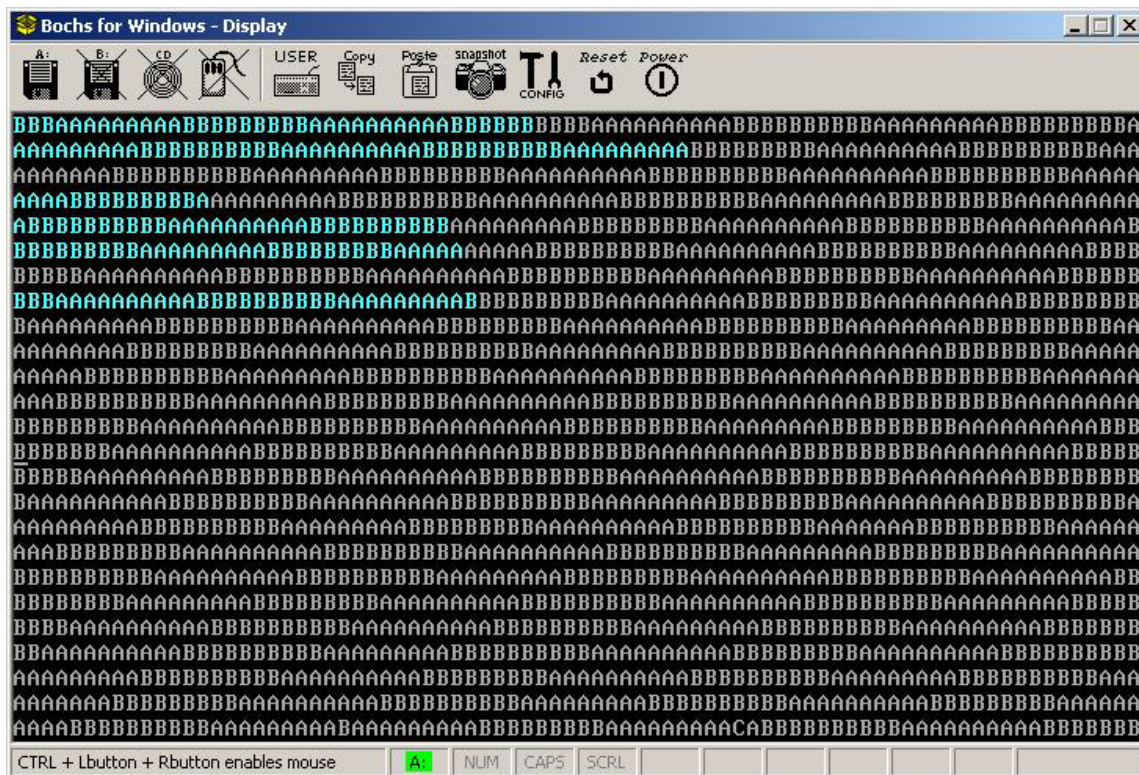


图 17-5 简单内核运行的屏幕显示情况

17.5 利用 Bochs 调试内核

Bochs 具有非常强大的操作系统内核调试功能。这也是本文选择 Bochs 作为首选实验环境的主要原因之一。有关 Bochs 调试功能的说明参见前面 14.2 节，这里基于 Linux 0.11 内核来说明 Windows 环境下 Bochs 系统调试操作的基本方法。

17.5.1 运行 Bochs 调试程序

我们假设 Bochs 系统已被安装在目录“C:\Program Files\Bochs-2.1.1\”中，并且 Linux 0.11 系统的 Bochs 配置文件名称是 bochsrc-hd.bxrc。现在我们在包含内核 Image 文件的目录下建立一个简单的批处理文件 run.bat，其内容如下：

```
"C:\Program Files\Bochs-2.1.1\bochsdbg" -q -f bochsrc-hd.bxrc
```

其中 bochsdbg 是 Bochs 系统的调试执行程序。运行该批处理命令即可进入调试环境。此时 Bochs 的主显示窗口空白，而控制窗口将显示以下类似内容：


```
C:\Documents and Settings\Linux-0.11>"C:\Program Files\Bochs-2.1.1\bochsdbg"
-q -f bochsrc-hd.bxrc
```

```
=====
                        Bochs x86 Emulator 2.1.1
                        February 08, 2004
=====
000000000000i[      ] reading configuration from bochsrc-hd.bxrc
000000000000i[      ] installing win32 module as the Bochs GUI
000000000000i[      ] Warning: no rc file specified.
000000000000i[      ] using log file bochsout.txt
Next at t=0
(0) context not implemented because BX_HAVE_HASH_MAP=0
[0x000ffff0] f000:ffff (unk. ctxt): jmp f000:e05b          ; ea5be000f0
<bochs:1>
```

此时 Bochs 调试系统已经准备好开始运行，CPU 执行指针已指向 ROM BIOS 中地址 0x000ffff0 处的指令处。其中 '<bochs:1>' 是命令输入行提示符，其中的数字表示当前的命令序列号。在命令提示符 '<bochs:1>' 后面键入 'help' 命令，可以列出调试系统的基本命令。若要了解某个命令的具体使用方法，可以键入 'help' 命令并且后面跟随一个用单引号括住的具体命令，例如：“help 'vbreak'”。见如下面所示。

```
<bochs:1> help
help - show list of debugger commands
help 'command' - show short command description
-* Debugger control -*-
    help, q|quit|exit, set, instrument, show, trace-on, trace-off,
    record, playback, load-symbols, slist
-* Execution control -*-
    c|cont, s|step|stepl, p|n|next, modebp
-* Breakpoint management -*-
    v|vbreak, lb|lbreak, pb|pbreak|b|break, sb, sba, blist,
    bpe, bpd, d|del|delete
-* CPU and memory contents -*-
    x, xp, u|disas|disassemble, r|reg|registers, setpmem, crc, info, dump_cpu,
    set_cpu, ptime, print-stack, watch, unwatch, ?|calc
<bochs:2> help 'vbreak'
help vbreak
vbreak seg:off - set a virtual address instruction breakpoint
<bochs:3>
```

以下是一些比较常用的命令。所有调试命令的完整列表请参见 Bochs 自带的 html 格式的帮助用户文件（internal-debugger.html）或者参考在线帮助信息（help 命令）。

1. 执行控制命令。控制指令的单步或多步执行。

c	连续执行
stepl [count]	执行 count 条指令，默认为 1 条。
si [count]	执行 count 条指令，默认为 1 条。
step [count]	执行 count 条指令，默认为 1 条。
s [count]	执行 count 条指令，默认为 1 条。
p	与 s 类似，但把中断指令和函数调用指令当作单步执行，即执行整个中断或子函数。
n(或 next)	与 s 类似，但把中断指令和函数调用指令当作单步执行，即执行整个中断或子函数。
Ctrl-C	停止执行，并回到命令行提示符下。

Ctrl-D	如果在空的命令行提示符下键入该命令，则退出 Bochs。
quit	退出调试和执行。
q	退出调试和执行。

2. 断点设置命令。其中 `seg`、`off` 和 `addr` 可以是'0x'开始的十六进制数，也可以是十进制数或者是以'0'开始的八进制数。

<code>vb</code>	<code>seg:off</code>	在虚拟地址上设置指令断点。
<code>lb</code>	<code>addr</code>	在线性地址上设置指令断点。
<code>pb</code>	<code>[*] addr</code>	在物理地址上设置指令断点。其中 '*' 是为了与 GDB 兼容的可选项。
<code>break</code>	<code>[*] addr</code>	
<code>b</code>	<code>[*] addr</code>	
<code>info break</code>		显示所有当前断点的状态。
<code>delete n</code>		删除一个断点。
<code>del n</code>		
<code>d n</code>		

3. 内存操作命令

<code>x /nuf addr</code>	检查位于线性地址 <code>addr</code> 处的内存内容，若 <code>addr</code> 不指定，则默认为下一个单元地址。
<code>xp /nuf addr</code>	检查位于物理地址 <code>addr</code> 处的内存内容。

其中的可选参数 `n`、`u` 和 `f` 的分别可为：

<code>n</code>	欲显示内存单元的计数值，默认值为 1。
<code>u</code>	表示单元大小，默认选择为 'w'：
<code>b</code> (Bytes)	1 字节；
<code>h</code> (Halfwords)	2 字节；
<code>w</code> (Words)	4 字节；
<code>g</code> (Giantwords)	8 字节。
注意：这些缩略符与 Intel 的不同，主要是为了与 GDB 调试器的表示法一致。	
<code>f</code>	显示格式，默认选择为 'x'：
<code>x</code> (hex)	显示为十六进制数（默认选择）；
<code>d</code> (decimal)	显示为十进制数；
<code>u</code> (unsigned)	显示成无符号十进制数；
<code>o</code> (octal)	显示成八进制数；
<code>t</code> (binary)	显示成二进制数。
<code>c</code> (char)	显示字节代码对应的字符。若不是可显示字符代码，就直接显示代码。

<code>crc addr1 addr2</code>	显示物理内存从 <code>addr1</code> 到 <code>addr2</code> 范围内内存的 CRC 校验值。
<code>info dirty</code>	显示上一次执行本命令以来已被修改过的物理内存页面。仅显示页面的前 20 字节。

4. 信息显示和 CPU 寄存器操作命令

<code>info program</code>	显示程序的执行状态。
<code>info registers</code>	列表显示 CPU 整数寄存器（相对于浮点寄存器）及其内容。
<code>info break</code>	显示当前断点设置状态信息。
<code>set \$reg = val</code>	修改 CPU 某一寄存器内容。目前除段寄存器和标志寄存器以外的寄存器都可以修改。 例如， <code>set \$eax = 0x01234567</code> ； <code>set \$edx = 25</code>
<code>dump_cpu</code>	显示 CPU 全部状态信息。
<code>set_cpu</code>	设置 CPU 全部状态信息。

"dump_cpu" 和 "set_cpu" 命令格式为:

```

"eax:0x%x\n"
"ebx:0x%x\n"
"ecx:0x%x\n"
"edx:0x%x\n"
"ebp:0x%x\n"
"esi:0x%x\n"
"edi:0x%x\n"
"esp:0x%x\n"
"eflags:0x%x\n"
"eip:0x%x\n"
"cs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"ss:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"ds:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"es:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"fs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"gs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"ldtr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"tr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
"gdtr:base=0x%x, limit=0x%x\n"
"idtr:base=0x%x, limit=0x%x\n"
"dr0:0x%x\n"
"dr1:0x%x\n"
"dr2:0x%x\n"
"dr3:0x%x\n"
"dr4:0x%x\n"
"dr5:0x%x\n"
"dr6:0x%x\n"
"dr7:0x%x\n"
"tr3:0x%x\n"
"tr4:0x%x\n"
"tr5:0x%x\n"
"tr6:0x%x\n"
"tr7:0x%x\n"
"cr0:0x%x\n"
"cr1:0x%x\n"
"cr2:0x%x\n"
"cr3:0x%x\n"
"cr4:0x%x\n"
"inhibit_int:%u\n"
"done\n"

```

其中:

- s (Selector) 是选择符;
- dl (Descriptor Low-dword) 是段描述符在选择符影子寄存器中的低 4 字节值;
- dh (Descriptor High-dword) 是段描述符在选择符影子寄存器中的高 4 字节值;
- valid 表示段寄存器中是否正存放着有效影子描述符。
- inhibit_int 是一个指令延迟中断标志。若置位, 则表示前一条刚执行过的指令是一条推迟 CPU 接受中断的指令 (例如 STI、MOV SS);

另外, 执行"set_cpu"命令出现任何错误时会使用格式 "Error: ..."报告出错信息。这些出错信息可能出现在每条输入行后面, 也可能出现在最后显示 "done"之后。若使用成功执行了"set_cpu"命令, 则该命

令将会显示“OK”来结束命令。

4. 反汇编命令

```
disassemble start end    对给定线性地址范围内的指令进行反汇编。
disas
u
```

以下是 Bochs 的一些新命令，但在 windows 环境下涉及到文件名的命令可能不能正常使用。

- **record *filename*** 把执行过程中你的输入命令序列写到文件 *filename* 中。该文件将包含格式为“%s %d %x”的行。其中第 1 个参数是事件类型；第 2 个是时间戳；第 3 个是相关事件的数据。
- **playback *filename*** 使用文件 *filename* 中的内容回放命令执行。在控制窗口中还可以直接键入其他命令。文件中的各事件将被回放，各时间的回放时刻将相对于该命令执行的时间算起。
- **print-stack [num words]** 显示堆栈顶端 num 个 16 位的字。num 默认值是 16 个。当堆栈段的基地址是 0 时该命令仅在保护模式下可以正常的使用。
- **load-symbols [global] *filename* [offset]** 从文件 *filename* 中加载符号信息。如果给出了关键字 global，那么在符号未加载以前的上下文中所有符号也都将是可见的。偏移 offset（默认为 0）会加入到每个符号项中。符号信息是加载到当前执行代码的上下文中的。符号文件 *filename* 中每行的格式是“%x %s”。其中第 1 个值是地址，第 2 个是符号名。

为了让 Bochs 直接模拟执行到 Linux 的引导启动程序开始处，我们可以先使用断点命令在 0x7c00 处设置一个断点，然后让系统连续运行到 0x7c00 处停下来。执行的命令序列如下：

```
<bochs:3> vbreak 0x0000:0x7c00
<bochs:4> c
(0) Breakpoint 1, 0x7c00 (0x0:0x7c00)
Next at t=4409138
(0) [0x00007c00] 0000:7c00 (unk. ctxt): mov ax, 0x7c0          ; b8c007
<bochs:5>
```

此时，CPU 执行到 boot.s 程序开始处的第 1 条指令处，Bochs 主窗口将显示出“Boot From floppy...”等一些信息。现在，我们可以利用单步执行命令's'或'n'（不跟踪进入子程序）来跟踪调试程序了。在调试时可以使用 Bochs 的断点设置命令、反汇编命令、信息显示命令等来辅助我们的调试操作。下面是一些常用命令的示例：

```
<bochs:8> u /10                                # 反汇编从当前地址开始的 10 条指令。
00007c00: (          ): mov ax, 0x7c0          ; b8c007
00007c03: (          ): mov ds, ax            ; 8ed8
00007c05: (          ): mov ax, 0x9000        ; b80090
00007c08: (          ): mov es, ax            ; 8ec0
00007c0a: (          ): mov cx, 0x100         ; b90001
00007c0d: (          ): sub si, si            ; 29f6
00007c0f: (          ): sub di, di            ; 29ff
00007c11: (          ): rep movs word ptr [di], word ptr [si] ; f3a5
00007c13: (          ): jmp 9000:0018          ; ea18000090
00007c18: (          ): mov ax, cs            ; 8cc8
<bochs:9> info r                                # 查看当前 CPU 寄存器的内容
eax                0xaa55                43605
ecx                0x110001              1114113
```

```

edx          0x0          0
ebx          0x0          0
esp          0xffffe      0xffffe
ebp          0x0          0x0
esi          0x0          0
edi          0xffe4       65508
eip          0x7c00       0x7c00
eflags       0x282        642
cs           0x0          0
ss           0x0          0
ds           0x0          0
es           0x0          0
fs           0x0          0
gs           0x0          0

```

```
<bochs:10> print-stack # 显示当前堆栈的内容
```

```

0000ffff [0000ffff] 0000
00010000 [00010000] 0000
00010002 [00010002] 0000
00010004 [00010004] 0000
00010006 [00010006] 0000
00010008 [00010008] 0000
0001000a [0001000a] 0000

```

```
...
```

```
<bochs:11> dump_cpu
```

```
# 显示 CPU 中的所有寄存器和状态值。
```

```

eax:0xaa55
ebx:0x0
ecx:0x110001
edx:0x0
ebp:0x0
esi:0x0
edi:0xffe4
esp:0xffffe
eflags:0x282
eip:0x7c00
cs:s=0x0, dl=0xffff, dh=0x9b00, valid=1
ss:s=0x0, dl=0xffff, dh=0x9300, valid=7
ds:s=0x0, dl=0xffff, dh=0x9300, valid=1
es:s=0x0, dl=0xffff, dh=0x9300, valid=1
fs:s=0x0, dl=0xffff, dh=0x9300, valid=1
gs:s=0x0, dl=0xffff, dh=0x9300, valid=1
ldtr:s=0x0, dl=0x0, dh=0x0, valid=0
tr:s=0x0, dl=0x0, dh=0x0, valid=0
gdtr:base=0x0, limit=0x0
idtr:base=0x0, limit=0x3ff
dr0:0x0
dr1:0x0
dr2:0x0
dr3:0x0
dr6:0xffff0ff0
dr7:0x400
tr3:0x0
tr4:0x0
tr5:0x0

```

```
# s 是选择符；dl 和 dh 分别是描述符低、高双字。
```

```
tr6:0x0
tr7:0x0
cr0:0x60000010
cr1:0x0
cr2:0x0
cr3:0x0
cr4:0x0
inhibit_mask:0
done
<bochs:12>
```

由于 Linux 0.11 内核的 32 位代码是从绝对物理地址 0 处开始存放的，因此若想直接执行到 32 位代码开始处，即 `head.s` 程序开始处，我们可以在线性地址 0x0000 处设置一个断点并运行命令 `c` 执行到那个位置处。

另外，当直接在命令提示符下打回车键时会重复执行上一个命令；按向上方向键会显示上一命令。其他命令的使用方法请参考 `help` 命令。

17.5.2 定位内核中的变量或数据结构

在编译内核时会产生一个 `system.map` 文件。该文件列出了内核 `Image (bootimage)` 文件中全局变量和各个模块中的局部变量的偏移地址位置。在内核编译完成后可以使用前面介绍的文件导出方法把 `system.map` 文件抽取到主机环境（windows）中。有关 `system.map` 文件的详细功能和作用请参见 2.10.3 节。`system.map` 样例文件中的部分内容见如下所示。利用这个文件，我们可以在 Bochs 调试系统中快速地定位某个变量或跳转到指定的函数代码处。

```
...
Global symbols:

_dup: 0x16e2c
_nmi: 0x8e08
_bmap: 0xc364
_iput: 0xc3b4
_blk_dev_init: 0x10ed0
_open: 0x16dbc
_do_execve: 0xe3d4
_con_init: 0x15ccc
_put_super: 0xd394
_sys_setgid: 0x9b54
_sys_umask: 0x9f54
_con_write: 0x14f64
_show_task: 0x6a54
buffer_init: 0xd1ec
_sys_settimeofday: 0x9f4c
_sys_getgroups: 0x9edc
...
```

同样，由于 Linux 0.11 内核的 32 位代码是从绝对物理地址 0 处开始存放的，`system.map` 中全局变量的偏移位置值就是 CPU 中线性地址位置，因此我们可以直接在感兴趣的变量或函数名位置处设置断点，并让程序连续执行到指定的位置处。例如若我们想调试函数 `buffer_init()`，那么从 `system.map` 文件中可以

知道它位于 0xd1ec 处。此时我们可以在该处设置一个线性地址断点，并执行命令'c'让 CPU 执行到这个指定的函数开始处，见如下所示。

```

<bochs:12> lb 0xd1ec                                # 设置线性地址断点。
<bochs:13> c                                          # 连续执行。
(0) Breakpoint 2, 0xd1ec in ?? ()
Next at t=16689666
(0) [0x0000d1ec] 0008:0000d1ec (unk. ctxt): push ebx          ; 53
<bochs:14> n                                          # 执行下一指令。
Next at t=16689667
(0) [0x0000d1ed] 0008:0000d1ed (unk. ctxt): mov eax, dword ptr ss:[esp+0x8] ; 8b442408
<bochs:15> n                                          # 执行下一指令。
Next at t=16689668
(0) [0x0000d1f1] 0008:0000d1f1 (unk. ctxt): mov edx, dword ptr [ds:0x19958] ; 8b1558990100
<bochs:16>

```

程序调试是一种技能，需要多练习才能熟能生巧。上面介绍的一些基本命令需要组合在一起使用才能灵活地观察到内核代码执行的整体环境情况。

17.6 创建磁盘映像文件

磁盘映像文件（Disk Image File）是软盘或硬盘上信息的一个完整映像，并以文件的形式保存。磁盘映像文件中存储信息的格式与对应磁盘上保存信息的格式完全一样。空磁盘映像文件是容量与我们创建的磁盘相同但内容全为 0 的一个文件。这些空映像文件就象刚买来的新软盘或硬盘，还需要经过分区或/以及格式化才能使用。

在制作磁盘映像文件之前，我们首先需要确定所创建映像文件的容量。对于软盘映像文件，各种规格（1.2MB 或 1.44MB）的容量都是固定的。因此这里主要说明如何确定自己需要的硬盘映像文件的容量。普通硬盘的结构由堆积的金属圆盘组成。每个圆盘的上下两面用于保存数据，并且以同心圆的方式把整个表面划分成一个个磁道，或称为柱面（Cylinder）。因此一个圆盘需要一个磁头（Head）来读写上面的数据。在圆盘旋转时磁头只需要作径向移动就可以在任何磁道上方移动，从而能够访问圆盘表面所有有效的位置。每个磁道被划分成若干个扇区，扇区长度一般由 256 -- 1024 字节组成。对于大多数系统来说，通常扇区长度均为 512 字节。一个典型的硬盘结构见图 2-11 所示。

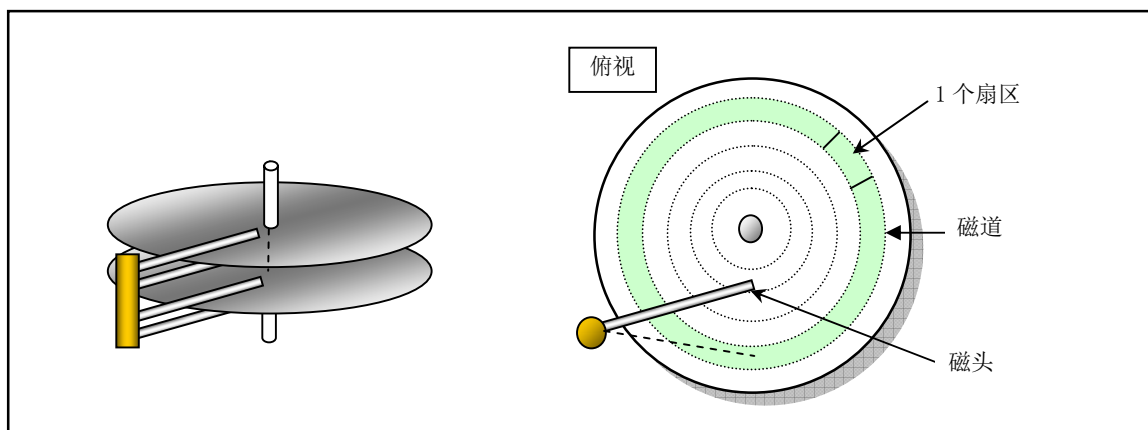


图 17-6 典型硬盘内部结构

图中示出了具有两个金属圆盘的硬盘结构。因此该硬盘有 4 个物理磁头。所含的最大柱面数在生产时已确定。当对硬盘进行分区和格式化时，圆盘表面的磁介质就被初始化成指定格式的数据，从而每个磁道（或柱面）被划分成指定数量的扇区。因此这个硬盘的总扇区数为：

$$\text{硬盘总扇区数} = \text{物理磁道数} \times \text{物理磁头数} \times \text{每磁道扇区数}$$

硬盘中以上这些实际的物理参数与一个操作系统中所使用的参数会有区别，称为逻辑参数。但这些参数所计算出的总扇区数与硬盘物理参数计算出的肯定是相同的。由于在设计 PC 机系统时没有考虑到硬件设备性能和容量发展得如此之快，ROM BIOS 某些表示硬盘参数所使用的比特位太少而不能符合实际硬盘物理参数的要求。因此目前操作系统或机器 BIOS 中普遍采用的措施就是在保证硬盘总扇区数相等的情况下适当调整磁道数、磁头数和每磁道扇区数，以符合兼容性和参数表示限制的要求。在 Bochs 配置文件有关硬盘设备参数中的变换（Translation）选项也是为此目的而设置的。

在我们为 Linux 0.11 系统制作硬盘 Image 文件时，考虑到其本身代码量很少，而且所使用的 MINIX 1.5 文件系统最大容量为 64MB 的限制，因此每个硬盘分区大小最大也只能是 64MB。另外，Linux 0.11 系统尚未支持扩展分区，因此对于一个硬盘 Image 文件来说，最多有 4 个分区。因此，Linux 0.11 系统可使用的硬盘 Image 文件最大容量是 $64 \times 4 = 256\text{MB}$ 。在下面的说明中，我们将以创建一个具有 4 个分区、每个分区为 60MB 的硬盘 Image 文件为例子进行说明。

对于软盘来说，我们可以把它看作是一种具有固定磁道数（柱面数）、磁头数和每磁道扇区数（spt - Sectors Per Track）的超小型硬盘。例如容量是 1.44MB 的软盘参数是 80 个磁道、2 个磁头和每磁道有 18 个扇区、每个扇区有 512 字节。其扇区总数是 2880，总容量是 $80 \times 2 \times 18 \times 512 = 1474560$ 字节。因此下面介绍的所有针对硬盘映像文件的制作方式都可以用来制作软盘映像文件。为了叙述上的方便，在没有特别指出时，我们把所有磁盘映像文件统称为 Image 文件。

17.6.1 利用 Bochs 软件自带的 Image 生成工具

Bochs 系统带有一个 Image 生成工具“Disk Image Creation Tool”（bximage.exe）。用它制作软盘和硬盘的空 Image 文件。在运行并出现了 Image 创建界面时，程序首先会提示选择需要创建的 Image 类型（硬盘 hd 还是软盘 fd）。若是创建硬盘，还会提示输入硬盘 Image 的 mode 类型。通常只需要选择其默认值 flat 即可。然后输入你需要创建的 Image 容量。程序会显示对应的硬盘参数值：柱面数（磁道数、磁头数和每磁道扇区数），并要求输入 Image 文件的名称。程序在生成了 Image 文件之后，会显示一条用于 Bochs 配置文件中设置硬盘参数的配置信息。记下这条信息并编辑到配置文件中。下面是创建一个 256MB 硬盘 Image 文件的过程。

```
=====
                        bximage
                Disk Image Creation Tool for Bochs
                $Id: bximage.c,v 1.19 2003/08/01 01:20:00 cbothamy Exp $
=====
Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd]

What kind of image should I create?
Please type flat, sparse or growing. [flat]

Enter the hard disk size in megabytes, between 1 and 32255
```

[10] 256

```
I will create a 'flat' hard disk image with
cyl=520
heads=16
sectors per track=63
total sectors=524160
total size=255.94 megabytes
```

What should I name the image?

```
[c.img] hdc.img
```

```
Writing: [] Done.
```

I wrote 268369920 bytes to (null).

The following line should appear in your bochsrc:

```
ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63
```

Press any key to continue

如果已经有了一个容量满足要求的硬盘 Image 文件，那么可以直接复制该文件就能产生另一个 Image 文件。然后可以按照自己的要求对该文件进行处理。对于创建软盘 Image 文件其过程与上述类似，只是还会提示你选择软盘种类的提示。同样，如果已经有其他软盘 Image 文件，那么采用直接复制方法即可。

17.6.2 在 Linux 系统下使用 dd 命令创建 Image 文件。

前面已经说明，刚创建的 Image 文件是一个内容全为 0 的空文件，只是其容量与要求的一致。因此我们可以首先计算出要求容量的 Image 文件的扇区数，然后使用 dd 命令来产生相应的 Image 文件。

例如我们想要建立柱面数是 520、磁头数是 16、每磁道扇区数是 63 的硬盘 Image 文件，其扇区总数为： $520 * 16 * 63 = 524160$ ，则命令为：

```
dd if=/dev/zero of=hdc.img bs=512 count=524160
```

对于 1.44MB 的软盘 Image 文件，其扇区数是 2880，因此命令为：

```
dd if=/dev/zero of=diska.img bs=512 count=2880
```

17.6.3 利用 WinImage 创建 DOS 格式的软盘 Image 文件

WinImage 是一个 DOS 格式 Image 文件访问和创建工具。双击 DOS 软盘 Image 文件的图标就可以浏览、删除或往里添加文件。除此之外，它还能用于浏览 CDROM 的 iso 文件。使用 WinImage 创建软盘 Image 时可以生成一个带有 DOS 格式的 Image 文件。方法如下：

- 运行 WinImage。选择“Options->Settings”菜单，选择其中的 Image 设置页。设置 Compression 为“None”（也即把指示标拉到最左边）。
- 创建 Image 文件。选择菜单 File->New，此时会弹出一个软盘格式选择框。请选择容量是 1.44MB 的格式。
- 再选择引导扇区属性菜单项 Image->Boot Sector properties，单击对话框中的 MS-DOS 按钮。
- 保存文件。

注意，在保存文件对话框中“保存类型”一定要选择“All files (*.*)”，否则创建的 Image 文件中会

包含一些 WinImage 自己的信息，从而会造成 Image 文件在 Bochs 下不能正常使用。可以通过查看文件长度来确定新创建 Image 是否符合要求。标准 1.44MB 软盘的容量应该是 1474560 字节。如果新的 Image 文件长度大于该值，那么请严格按照所述方法重新制作或者使用 UltraEdit 等二进制编辑器删除多余的字节。删除操作的方法如下：

- 使用 UltraEdit 以二进制模式打开 Image 文件。根据磁盘映像文件第 511, 512 字节是 55,AA 两个十六进制数，我们倒推 512 字节，删除这之前的所有字节。此时对于使用 MSDOS5.0 作为引导的磁盘来讲，文件头几个字节应该类似于“EB 3C 90 4D ...”。
- 然后下拉右边滚动条，移动到 img 文件末尾处。删除“...F6 F6 F6”后面的所有数据。通常来讲就是删除从 0x168000 开始的所有数据。操作完成时最后一行应该是完整的一行“F6 F6 F6...”。存盘退出即可使用该 Image 文件了。

17.7 制作根文件系统

本节的目标是在硬盘上建立一个根文件系统。虽然在 oldlinux.org 上可以下载到已经制作好的软盘和硬盘根文件系统 Image 文件，但这里还是把制作过程详细描述一遍，以供大家学习参考。在制作过程中还可以参考 Linus 的文章：INSTALL-0.11。在制作根文件系统盘之前，我们首先下载 rootimage-0.11 和 bootimage-0.11 映像文件（请下载日期最新的相关文件）：

<http://oldlinux.org/Linux.old/images/bootimage-0.11-20040305>

<http://oldlinux.org/Linux.old/images/rootimage-0.11-20040305>

将这两个文件修改成便于记忆的名称 bootimage-0.11 和 rootimage-0.11，并专门建立一个名为 Linux-0.11 的子目录。在制作过程中，我们需要复制 rootimage-0.11 软盘中的一些执行程序，并使用 bootimage-0.11 引导盘来启动模拟系统。因此在开始着手制作根文件系统之前，首先需要确认已经能够运行这两个软盘 Image 文件组成的最小 Linux 系统。

17.7.1 根文件系统和根文件设备

Linux 引导启动时，默认使用的文件系统是根文件系统。其中一般都包括以下一些子目录和文件：

- ♦ etc/ 目录主要含有一些系统配置文件；
- ♦ dev/ 含有设备特殊文件，用于使用文件操作语句操作设备；
- ♦ bin/ 存放系统执行程序。例如 sh、mkfs、fdisk 等；
- ♦ usr/ 存放库函数、手册和其他一些文件；
- ♦ usr/bin 存放用户常用的普通命令；
- ♦ var/ 用于存放系统运行时可变的数据或者是日志等信息。

存放文件系统的设备就是文件系统设备。比如，对于一般使用的 Windows2000 操作系统，硬盘 C 盘就是文件系统设备，而硬盘上按一定规则存放的文件就组成文件系统，Windows2000 有 NTFS 或 FAT32 等文件系统。而 Linux 0.11 内核所支持的文件系统是 MINIX 1.0 文件系统。

当 Linux 启动盘加载根文件系统时，会根据启动盘上引导扇区第 509、510 字节处一个字（ROOT_DEV）中的根文件系统设备号从指定的设备中加载根文件系统。如果这个设备号是 0 的话，则表示需要从引导盘所在当前驱动器中加载根文件系统。若该设备号是一个硬盘分区设备号的话，就会从该指定硬盘分区中加载根文件系统。Linux 0.11 内核中支持的硬盘设备号见表 17-3 所示。若该设备号是

一个软盘驱动器设备号的话，内核就会从该设备号指定的软驱中加载根文件系统。Linux 0.11 内核中使用的软盘驱动器设备号见表 17-4 所示。软盘驱动器设备号的计算方法请参见第 6 章 floppy.c 程序后的说明。

表 17-3 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

表 17-4 软盘驱动器逻辑设备号

逻辑设备号	对应设备文件	说明
0x0208	/dev/at0	1.2MB A 驱动器
0x0209	/dev/at1	1.2MB B 驱动器
0x021c	/dev/fd0	1.44MB A 驱动器
0x021d	/dev/fd1	1.44MB B 驱动器

17.7.2 创建文件系统

对于上面创建的硬盘 Image 文件，在能使用之前还必须对其进行分区和创建文件系统。通常的做法是把需要处理的硬盘 Image 文件挂接到 Bochs 下已有的模拟系统中（例如上面提到的 SLS Linux），然后使用模拟系统中的命令对新的 Image 文件进行处理。下面假设你已经安装了 SLS Linux 模拟系统，并且该系统存放在名称为 SLS-Linux 的子目录中。我们利用它对上面创建的 256MB 硬盘 Image 文件 hdc.img 进行分区并创建 MINIX 文件系统。我们将在这个 Image 文件中创建 1 个分区，并且建立成 MINIX 文件系统。我们执行的步骤如下：

1. 在 SLS-Linux 同级目录下建立一个名称为 Linux-0.11 的子目录，把 hdc.img 文件移动到该目录下。
2. 进入 SLS-Linux 目录，编辑 SLS Linux 系统的 Bochs 配置文件 bochsrc.bxrc。在 ata0-master 一行下加入我们的硬盘 Image 文件的配置参数行：
`ata0-slave:type=disk, path=..\Linux-0.11\hdc.img, cylinders=520, heads=16, spt=63`
3. 退出编辑器。双击 bochsrc.bxrc 的图标，运行 SLS Linux 模拟系统。在出现 Login 提示符时键入 'root' 并按回车键。如果此时 Bochs 不能正常运行，一般是由于配置文件信息有误，请重新编辑该配置文件。
4. 利用 fdisk 命令在 hdc.img 文件中建立 1 个分区。下面是建立第 1 个分区的命令序列。建立另外 3 个分区的过程与此相仿。由于 SLS Linux 默认建立的分区类型是支持 MINIX2.0 文件系统的 81 类型（Linux/MINIX），因此需要使用 fdisk 的 t 命令把类型修改成 80（Old MINIX）类型。这里请注意，

我们已经把 `hdc.img` 挂接成 SLS Linux 系统下的第 2 个硬盘。按照 Linux 0.11 对硬盘的命名规则，该硬盘整体的设备名应为 `/dev/hd5`。但是从 Linux 0.95 版开始硬盘的命名规则已经修改成目前使用的规则，因此在 SLS Linux 下第 2 个硬盘整体的设备名称是 `/dev/hdb`。

```
[/]# fdisk /dev/hdb
Command (m for help): n
Command action
    e   extended
    p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-520): 1
Last cylinder or +size or +sizeM or +sizeK (1-520): +63M

Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): L
    0 Empty          8 AIX          75 PC/IX          b8 BSDI swap
    1 DOS 12-bit FAT  9 AIX bootable  80 Old MINIX      c7 Syrix
    2 XENIX root      a OPUS         81 Linux/MINIX    db CP/M
    3 XENIX user      40 Venix       82 Linux swap     e1 DOS access
    4 DOS 16-bit <32M 51 Novell?      83 Linux extfs    e3 DOS R/O
    5 Extended        52 Microport   93 Amoeba         f2 DOS secondary
    6 DOS 16-bit >=32 63 GNU HURD   94 Amoeba BBT     ff BBT
    7 OS/2 HPFS       64 Novell      b7 BSDI fs

Hex code (type L to list codes): 80

Command (m for help): p
Disk /dev/hdb: 16 heads, 63 sectors, 520 cylinders
Units = cylinders of 1008 * 512 bytes
   Device Boot  Begin    Start    End  Blocks   Id  System
/dev/hdb1          1         1    129   65015+   80  Old MINIX

Command (m for help): w
The partition table has been altered.
Please reboot before doing anything else.
[/]#
```

5. 请记住该分区中数据块数大小（这里是 65015），在创建文件系统时会使用到这个值。当分区建立好后，按照通常的做法需要重新启动一次系统，以让 SLS Linux 系统内核能正确识别这个新加的分区。
6. 再次进入 SLS Linux 模拟系统后，我们使用 `mkfs` 命令在刚建立的第 1 个分区上创建 MINIX 文件系统。命令与信息如下所示。这里创建了具有 64000 个数据块的分区（一个数据块为 1KB 字节）。

```
[/]# mkfs /dev/hdb1 64000
21333 inodes
64000 blocks
Firstdatazone=680 (680)
Zonesize=1024
Maxsize=268966912
[/]#
```

至此，我们完成了在 hdc.img 文件的第 1 个分区中创建文件系统的工作。当然，建立创建文件系统也可以在运行 Linux 0.11 软盘上的根文件系统时建立。的现在我们可以把这个分区中建立成一个根文件系统。

17.7.3 Linux-0.11 的 Bochs 配置文件

在 Bochs 模拟系统中运行 Linux 0.11 时，其配置文件 bochsrc.bxrc 中通常需要设置以下内容。

```
romimage: file=$BXSHARE\BIOS-bochs-latest, address=0xf0000
megs: 16
vgaromimage: $BXSHARE\VGABIOS-elpin-2.40
floppya: 1_44="bootimage-0.11", status=inserted
ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63
boot: a
log: bochsout.txt
panic: action=ask
#error: action=report
#info: action=report
#debug: action=ignore
ips: 1000000
mouse: enabled=0
```

我们可以把 SLS Linux 的 Bochs 配置文件 bochsrc.bxrc 复制到 Linux-0.11 目录中，然后修改成与上面相同的内容。需要特别注意 floppya、ata0-master 和 boot，这 3 个参数一定要与上面一致。

现在我们用鼠标双击这个配置文件。首先 Bochs 显示窗口应该出现图 17-7 中画面。

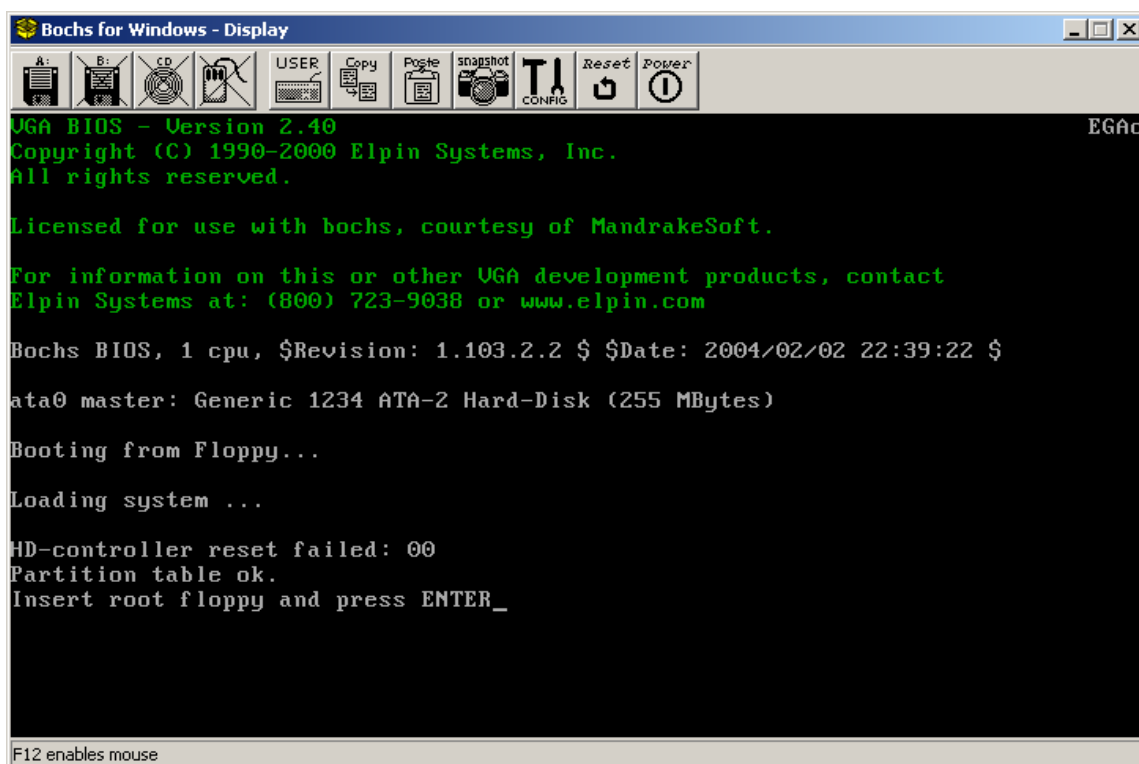


图 17-7 Bochs 系统运行窗口

此时应该单击窗口菜单条上 A:软盘图标，在对话框中把 A 盘配置为 rootimage-0.11 文件。或者采用 Bochs 配置窗口来设置。方法是单击菜单条上的'CONFIG'图标进入 Bochs 设置窗口（需要用鼠标点击才能把该窗口提到最前面），此时设置窗口显示的内容见图 17-8 所示。

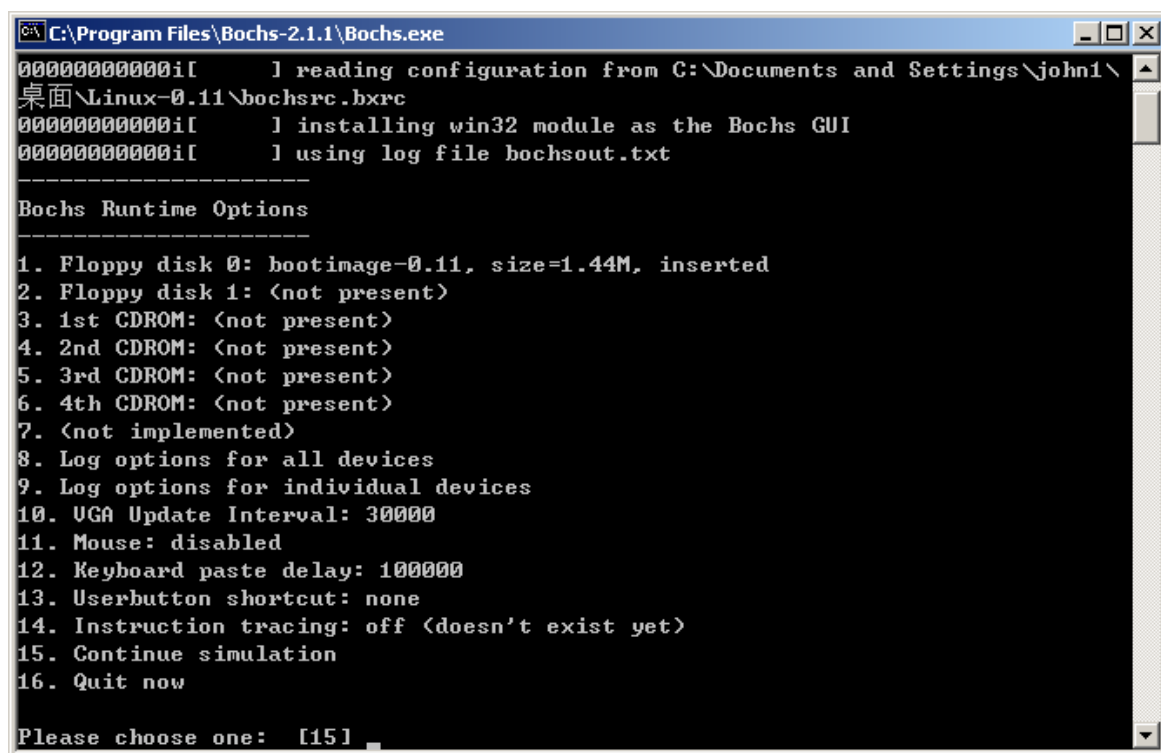


图 17-8 Bochs 系统配置窗口

修改其中第 1 项的软盘设置，让其指向 rootimage-0.11 盘。然后连续按回车键，直到设置窗口最后一行信息显示'Continuing simulation'为止。此时再切换到 Bochs 运行窗口。单击回车键后就正式进入了 Linux 0.11 系统。见图 17-9 所示。

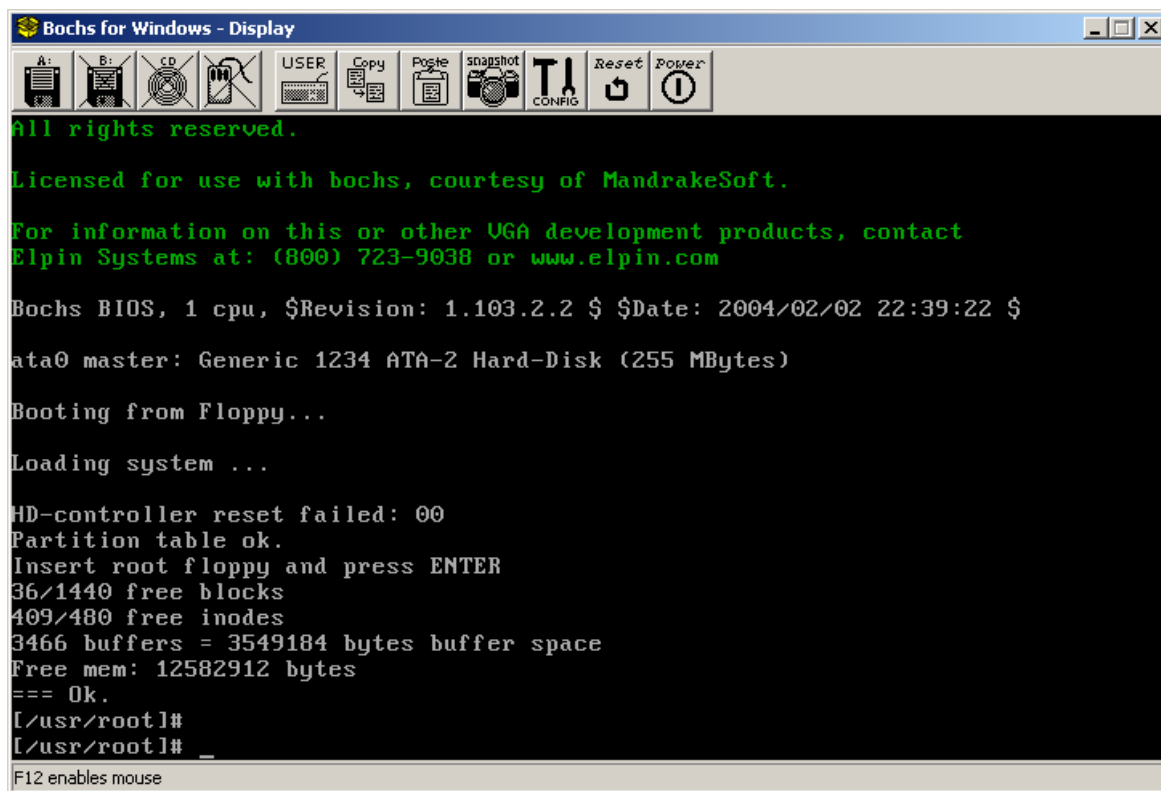


图 17-9 Bochs 中运行的 Linux 0.11 系统

17.7.4 在 hdc.img 上建立根文件系统

由于软盘容量太小,若要让 Linux 0.11 系统真正能做点什么的,就需要在硬盘(这里是指硬盘 Image 文件)上建立根文件系统。在前面我们已经建立一个 256MB 的硬盘 Image 文件 `hdc.img`, 并且此时已经连接到了运行着的 Bochs 环境中, 因此上图中出现一条有关硬盘的信息:

“ata0 master: Generic 1234 ATA-2 Hard-Disk (255 Mbytes)”

如果没有看到这条信息,说明你的 Linux 0.11 配置文件没有设置正确。请重新编辑 `bochsrc.bxrc` 文件, 并重新运行 Bochs 系统, 直到出现上述相同画面。

我们在前面已经在 `hdc.img` 第 1 个分区上建立了 MINIX 文件系统。若还没建好或者想再试一边的话, 那么就请键入一下命令来建立一个 64MB 的文件系统:

```
[/usr/root]# mkfs /dev/hd1 64000
```

现在可以开始加载硬盘上的文件系统了。执行下列命令, 把新的文件系统加载到 `/mnt` 目录上。

```
[/usr/root]# cd /
[/]# mount /dev/hd1 /mnt
[/]#
```

在加载了硬盘分区上的文件系统之后, 我们就可以把软盘上的根文件系统复制到硬盘上去了。请执

行以下命令：

```
[/]# cd /mnt
[/mnt]# for i in bin dev etc usr tmp
> do
> cp +recursive +verbose /$i $i
> done
```

此时软盘根文件系统上的所有文件就会被复制到硬盘上的文件系统中。在复制过程中会出现很多类似下面的信息。

```
/usr/bin/mv -> usr/bin/mv
/usr/bin/rm -> usr/bin/rm
/usr/bin/rmdir -> usr/bin/rmdir
/usr/bin/tail -> usr/bin/tail
/usr/bin/more -> usr/bin/more
/usr/local -> usr/local
/usr/root -> usr/root
/usr/root/.bash_history -> usr/root/.bash_history
/usr/root/a.out -> usr/root/a.out
/usr/root/hello.c -> usr/root/hello.c
/tmp -> tmp
[/mnt]# _
```

现在说明你已经在硬盘上建立好了一个基本的根文件系统。你可以在新文件系统中随处查看一下。然后卸载硬盘文件系统，并键入'logout'或'exit'退出 Linux 0.11 系统。此时会显示如下信息：

```
[/mnt]# cd /
[/]# umount /dev/hd1
[/]# logout
```

```
child 4 died with code 0000
[/usr/root]# _
```

17.7.5 使用硬盘 Image 上的根文件系统

一旦你在硬盘 Image 文件上建立好文件系统，就可以让 Linux 0.11 以它作为根文件系统启动。这通过修改引导盘 bootimage-0.11 文件的第 509、510 字节（0x1fc、0x1fd）的内容就可以实现。请按照以下步骤来进行。

1. 首先复制 bootimage-0.11 和 bochssrc.bxrc 两个文件，产生 bootimage-0.11-hd 和 bochssrc-hd.bxrc 文件。
2. 编辑 bochssrc-hd.bxrc 配置文件。把其中的'floppya:'上的文件名修改成'bootimage-0.11-hd'，并存盘。
3. 用 UltraEdit 或任何其他可修改二进制文件的编辑器（winhex 等）编辑 bootimage-0.11-hd 二进制文件。修改第 509、510 字节（即 0x1fc、0x1fd 处。原值应该是 00、00）为 01、03，表示根文件系统设备在硬盘 Image 的第 1 个分区上。然后存盘退出。如果把文件系统安装在了别的分区上，那么需要修改前 1 个字节以对应到你的分区上。

```
000001f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 03 55 AA; .....U?
```

现在可以双击 bochsrc-hd.bxrc 配置文件的图标，Bochs 系统应该会快速进入 Linux 0.11 系统并显示出图 17-10 中图形来。

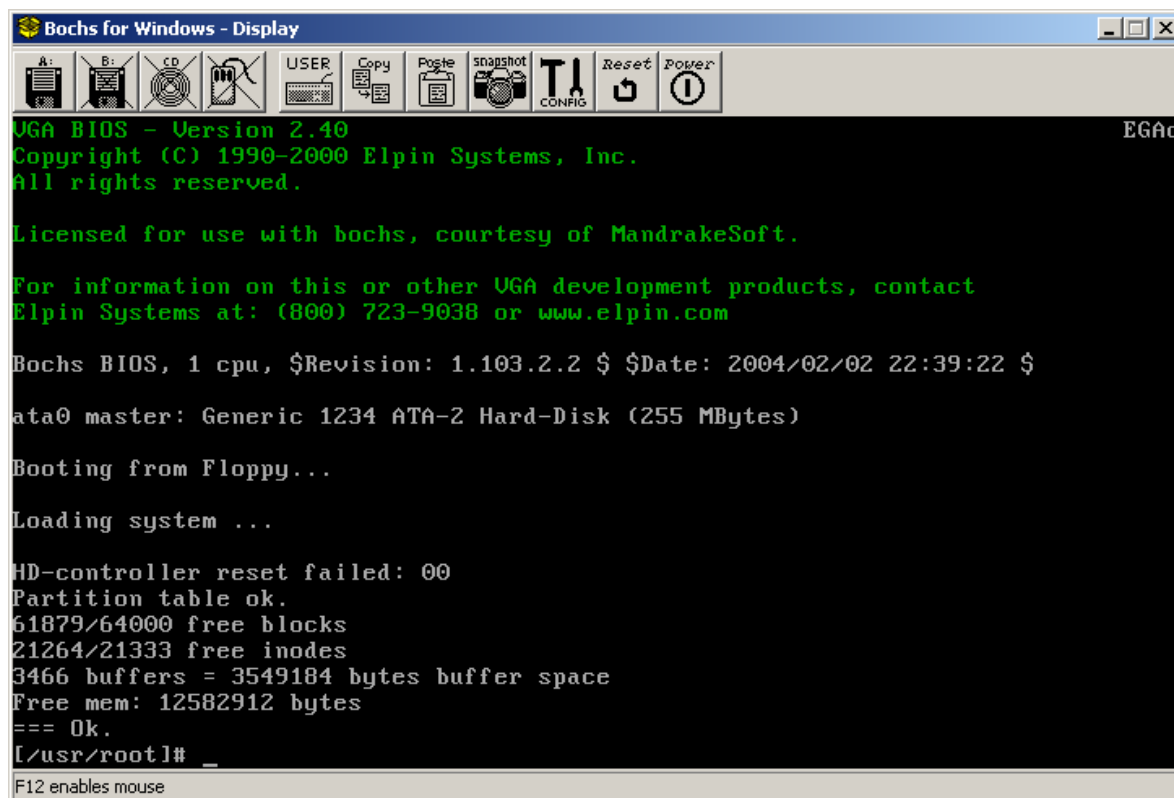


图 17-10 使用硬盘 Image 文件上的文件系统

17.8 在 Linux 0.11 系统上编译 0.11 内核

目前作者已经重新组建了一个带有 gcc 1.40 编译环境的 Linux 0.11 系统软件包。该系统设置成在 Bochs 仿真系统下运行，并且已经配置好相应的 bochs 配置文件。该软件包可从下面地址得到。

<http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040817.zip> 或
<http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040923.zip>

该软件包中含有一个 README 文件，其中说明了软件包中所有文件的作用和使用方法。若你的系统中已经安装了 bochs 系统，那么只需双击配置文件 bochs-hd.bxrc 的图标即可运行硬盘 Image 文件作为根文件系统的 Linux 0.11。在 /usr/src/linux 目录下键入 'make' 命令即可编译 Linux 0.11 内核源代码，并生成引导启动映像文件 Image。若需要输出这个 Image 文件，可以首先备份 bootimage-0.11-hd 文件，然后使用下面命令就会把 bootimage-0.11-hd 替换成新的引导启动文件。直接重新启动 Bochs 即可使用该新编译生成的 bootimage-0.11-hd 来引导系统。

```
[/usr/src/linux]# make
[/usr/src/linux]# dd bs=8192 if=Image of=/dev/fd0
[/usr/src/linux]# _
```

也可以使用 `mttools` 命令把新生成的 `Image` 文件写到第 2 个软盘映像文件 `diskb.img` 中，然后使用工具软件 `WinImage` 将 `diskb.img` 中的 `'Image'` 文件取出。

```
[/usr/src/linux]# mcopy Image b:
Copying IMAGE
[/usr/src/linux]# mcopy System.map b:
Copying SYSTEM.MAP
[/usr/src/linux]# mdir b:
Volume in drive B is B.
Directory for B:/
GCCLIB-1 TAZ      934577    3-29-104    7:49p
IMAGE            121344    4-29-104   11:46p
SYSTEM  MAP      17162    4-29-104   11:47p
README           764     3-29-104    8:03p
      4 File(s)      382976 bytes free
[/usr/src/linux]# _
```

如果想把新的引导启动 `Image` 文件与软盘上的根文件系统 `rootimage-0.11` 一起使用，那么在编译之前首先编辑 `Makefile` 文件，使用#注释掉 `'ROOT_DEV='` 一行内容即可。

在编译内核时通常可以很顺利地地完成。可能出现的问题是编译器 `gcc` 不能识别选项 `'-mstring-ins'`，这个选项是 `Linus` 对自己编译的 `gcc 1.40` 编译器做的扩展实验参数，用于对 `gcc` 生成字符串指令时进行优化处理。为了解决这个问题，可以直接删除所有 `Makefile` 中的这个参数再重新编译内核。另一个可能出现的问题是找不到 `gar` 命令，此时可以把 `/usr/local/bin/` 下的 `ar` 直接链接或复制/改名成 `gar` 即可。

17.9 在 Redhat 9 系统下编译 Linux 0.11 内核

最初的 `Linux` 操作系统内核是在 `Minix 1.5.10` 操作系统的扩展版本 `Minix-i386` 上交叉编译开发的。`Minix 1.5.10` 该版本的操作系统是随 `A.S. Tanenbaum` 的《`Minix 设计与实现`》一书第 1 版一起由 `Prentice Hall` 发售的。该版本的 `Minix` 虽然可以运行在 `80386` 及其兼容微机上，但并没有利用 `80386` 的 32 位机制。为了能在该系统上进行 32 位操作系统的开发，`Linus` 使用了 `Bruce Evans` 的补丁程序将其升级为 `MINIX-386`，并把 `GNU` 的系列开发工具 `gcc`、`gld`、`emacs`、`bash` 等移植到 `Minix-386` 上。在这个平台上，`Linus` 进行交叉编译，开发出 `Linux 0.01`、`0.03`、`0.11` 等版本的内核。作者曾根据 `Linux` 邮件列表中的文章介绍，建立起了类似 `Linus` 当时的开发平台，并顺利地编译出 `Linux` 的早期版本内核（见 <http://oldlinux.org> 论坛中的介绍）。

但由于 `Minix 1.5.10` 早已过时，而且该开发平台的建立非常烦琐，因此这里只简单介绍一下如何修改 `Linux 0.11` 版内核源代码，使其能在目前常用的 `RedHat 9` 操作系统标准的编译环境下进行编译，生成可运行的启动映像文件 `bootimage`。读者可以在普通 `PC` 机上或 `Bochs` 等虚拟机软件中运行它。这里仅给出主要的修改方面，所有的修改之处可使用工具 `diff` 来比较修改后和未修改前的代码，找出其中的区别。假如，未修改过的代码在 `linux` 目录中，修改过的代码在 `linux-mdf` 中，则需要执行下面的命令：

```
diff -r linux linux-mdf > dif.out
```

其中文件 dif.out 中即包含代码中所有修改过的地方。已经修改好并能在 RedHat 9 下编译的 Linux 0.11 内核源代码可以从下面地址处下载：

<http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.tar.gz>

<http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.diff.gz>

用编译好的启动映像文件软盘启动时，屏幕上应该会显示以下信息：

Booting from Floppy...

Loading system ...

Insert root floppy and press ENTER

如果在显示出“Loading system...”后就没有反应了，这说明内核不能识别计算机中的硬盘控制器子系统。可以找一台老式的 PC 机再试试，或者使用 vmware、bochs 等虚拟机软件试验。在要求插入根文件系统盘时，如果直接按回车键，则会显示以下不能加载根文件系统的信息，并且死机。若要完整地运行 linux 0.11 操作系统，则还需要与之相配的根文件系统，可以到 oldlinux.org 网站上下载一个使用。

<http://oldlinux.org/Linux.old/images/rootimage-0.11-for-orig>

17.9.1 修改 makefile 文件

在 Linux 0.11 内核代码文件中，几乎每个子目录中都包括一个 makefile 文件，需要对它们都进行以下修改：

- 将 gas=>as, gld=>ld。现在 gas 和 gld 已经直接改名称为 as 和 ld 了。
- as(原 gas)已经不用 -c 选项，因此需要去掉其 -c 编译选项。在内核主目录 Linux 下 Makefile 文件中，是在 34 行上。
- 去掉 gcc 的编译标志选项：-fcombine-regs、-mstring-insns 以及所有子目录中 Makefile 中的这两个选项。在 94 年的 gcc 手册中就已找不到 -fcombine-regs 选项，而 -mstring-insns 是 Linus 自己对 gcc 的修改增加的选项，所以你我的 gcc 中肯定不包括这个优化选项。
- 在 gcc 的编译标志选项中，增加 -m386 选项。这样在 RedHat 9 下编译出的内核映像文件中就不含有 80486 及以上 CPU 的指令，因此该内核就可以运行在 80386 机器上。

17.9.2 修改汇编程序中的注释

as86 编译程序不能识别 c 语言的注释语句，因此需要使用 ! 注释掉 boot/bootsect.s 文件中的 C 注释语句。

17.9.3 内存位置对齐语句 align 值的修改

在 boot 目录下的三个汇编程序中，align 语句使用的方法目前已经改变。原来 align 后面带的数值是指对起内存位置的幂次值，而现在则需要直接给出对起的整数地址值。因此，原来的语句：

```
.align 3
```

需要修改成(2 的 3 次幂值 $2^3=8$):

```
.align 8
```

17.9.4 修改嵌入宏汇编程序

由于对 `as` 的不断改进，目前其自动化程度越来越高，因此已经不需要人工指定一个变量需使用的 CPU 寄存器。因此内核代码中的 `__asm__ ("ax")` 需要全部去掉。例如 `fs/bitmap.c` 文件的第 20 行、26 行上，`fs/namei.c` 文件的第 65 行上等。

在嵌入汇编代码中，另外还需要去掉所有对寄存器内容无效（会被修改的寄存器）的声明。例如 `include/string.h` 中第 84 行：

```
:"si","di","ax","cx");
```

需要修改成如下的样子：

```
);
```

这样修改有时也会出现一些问题。由于 `gcc` 有时会根据上述声明对程序进行优化处理，某些地方若删除会被修改的寄存器内容就会造成 `gcc` 优化错误。因此程序代码中的某些地方还需要根据具体情况保留一些这类声明。例如在 `include/string.h` 文件 `memcpy()` 定义中的第 342 行。

17.9.5 c 程序变量在汇编语句中的引用表示

在开发 Linux 0.11 时所用的汇编器，在引用 C 程序中的变量时需要在变量名前加一下划线字符 '_'，而目前的 `gcc` 编译器可以直接识别使用这些汇编中引用的 c 变量，因此需要将汇编程序（包括嵌入汇编语句）中所有 c 变量之前的下划线去掉。例如 `boot/head.s` 程序中第 15 行语句：

```
.globl _idt,_gdt,_pg_dir,_tmp_floppy_area
```

需要改成：

```
.globl idt,gdt,pg_dir,tmp_floppy_area
```

第 31 行语句：

```
lss _stack_start,%esp
```

需要改成：

```
lss stack_start,%esp
```

17.9.6 保护模式下调试显示函数

在进入保护模式之前，可以用 ROM BIOS 中的 `int 0x10` 调用在屏幕上显示信息，但进入了保护模式后，这些中断调用就不能使用了。为了能在保护模式运行环境中了解内核的内部数据结构和状态，我们可以使用下面这个数据显示函数 `check_data32()`¹³。内核中虽然有 `printk()` 显示函数，但是它需要调用 `tty_write()`，在内核没有完全运转起来该函数是不能使用的。这个 `check_data32()` 函数可以在进入保护模式后，在屏幕上打印你感兴趣的东西。起用页功能与否，不影响效果，因为虚拟内存存在 4M 之内，正好使用了第一个页表目录项，而页表目录从物理地址 0 开始，再加上内核数据段基地址为 0，所以 4M 范围内，虚拟内存与线性内存以及物理内存的地址相同。`linux` 当初可能也这样斟酌过的，觉得这样设置使用起来比较方便☺。

嵌入式汇编语句的使用方法请参见第 3 章内容。

```
/*
```

```
* 作用：在屏幕上用 16 进制显示一个 32 位整数。
```

```
* 参数：value  -- 要显示的整数。
```

```
*      pos    -- 屏幕位置，以 16 个字符宽度为单位，例如为 2，即表示从左上角 32 字符宽度处开始显示。
```

```
* 返回：无。
```

```
* 如果要在汇编程序中用，要保证该函数被编译链接进了内核。gcc 汇编中的用法如下：
```

¹³ 该函数由 `oldlinux.org` 论坛上的朋友 `notrump` 提供。

```

* pushl pos      //pos 要用你实际的数据代替, 例如 pushl $4
* pushl value    //pos 和 value 可以是任何合法的寻址方式
* call  check_data32
*/
inline void check_data32(int value, int pos)
{
    __asm__ __volatile__(
        "shl    $4, %%ebx\n\t"           // 将 pos 值乘 16, 在加上 VGA 显示内存起始地址,
        "addl   $0xb8000, %%ebx\n\t"     // ebx 中得到在屏幕左上角开始的显示字符位置。
        "movl   $0xf0000000, %%eax\n\t"  // 设置 4 比特屏蔽码。
        "movb   $28, %%cl\n\t"          // 设置初始右移比特数值。
        "1:\n\t"
        "movl   %0, %%edx\n\t"           // 取欲显示的值 value→edx
        "andl   %%eax, %%edx\n\t"         // 取 edx 中有 eax 指定的 4 个比特。
        "shr    %%cl, %%edx\n\t"         // 右移 28 位, edx 中即为所取 4 比特的值。
        "add    $0x30, %%dx\n\t"         // 将该值转换成 ASCII 码。
        "cmp    $0x3a, %%dx\n\t"         // 若该 4 比特数值小于 10, 则向前跳转到标号 2 处。
        "jb2f\n\t"
        "add    $0x07, %%dx\n\t"         // 否则再加上 7, 将值转换成对应字符 A—F。
        "2:\n\t"
        "add    $0x0c00, %%dx\n\t"       // 设置显示属性。
        "movw   %%dx, (%%ebx)\n\t"       // 将该值放到显示内存中。
        "sub    $0x04, %%cl\n\t"         // 准备显示下一个 16 进制数, 右移比特位数减 4。
        "shr    $0x04, %%eax\n\t"       // 比特位屏蔽码右移 4 位。
        "add    $0x02, %%ebx\n\t"       // 更新显示内存位置。
        "cmpl   $0x0, %%eax\n\t"         // 屏蔽码值已经移出右端 (已经显示完 8 个 16 进制数)?
        "jnz1b\n\t"                     // 还有数值需要显示, 则向后跳转到标号 1 处。
        ::"m"(value), "b"(pos));
    }

```

17.10 内核引导启动+根文件系统组成的集成盘

本节内容主要说明制作由内核引导启动映像文件和根文件系统组合成的集成盘映像文件的制作原理和方法。主要目的是了解 Linux 0.11 内核内存虚拟盘工作原理, 并进一步理解引导盘和根文件系统盘的概念。加深对 kernel/blk_drv/ramdisk.c 程序运行方式的理解。在制作这个集成盘之前, 我们需要首先下载或准备好以下实验软件:

<http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040923.zip>

<http://oldlinux.org/Linux.old/images/rootimage-0.11-for-orig>

linux-0.11-devel 是运行在 Bochs 下的带开发环境的 Linux 0.11 系统, rootimage-0.11 是 1.44MB 软盘映像文件中的 Linux 0.11 根文件系统。后缀'for-orig'是指该根文件系统适用于未经修改的 Linux 0.11 内核源代码编译出的内核引导启动映像文件。当然这里所说的“未经修改”是指没有对内核作过什么大的改动, 因为我们还是要修改编译配置文件 Makefile, 以编译生成含有内存虚拟盘的内核代码来。

17.10.1 集成盘制作原理

通常我们使用软盘启动 Linux 0.11 系统时需要两张盘（这里“盘”均指对应软盘的 Image 文件）：一张是内核引导启动盘，一张是基本的根文件系统盘。这样必须使用两张盘才能引导启动系统来正常运行一个基本的 Linux 系统，并且在运行过程中根文件系统盘必须一直保持在软盘驱动器中。而我们这里描述的集成盘是指把内核引导启动盘和一个基本的根文件系统盘的内容合成制作在一张盘上。这样我们使用一张集成盘就能引导启动 Linux 0.11 系统到命令提示符状态。集成盘实际上就是一张含有根文件系统的内核引导盘。

为了能运行集成盘系统，该盘上的内核代码中需要开启内存虚拟盘（RAMDISK）的功能。这样集成盘上的根文件系统就能被加载到内存中的虚拟盘中，从而系统上的两个软盘驱动器就能腾出来用于加载（mount）其他文件系统盘或派其他用途。下面我们再详细介绍一下在一张 1.44MB 盘上制作成集成盘的原理和步骤。

17.10.1.1 引导过程原理

Linux 0.11 的内核在初始化时会根据编译时 Makefile 文件中设置的 RAMDISK 选项判断在系统物理内存是否要开辟虚拟盘区域。如果没有设置 RAMDISK（即其长度为 0）则内核会根据 ROOT_DEV 所设置的根文件系统所在设备号，从软盘或硬盘上加载根文件系统，执行无虚拟盘时的一般启动过程。

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。这个检测和加载过程参见图 17-11 所示。

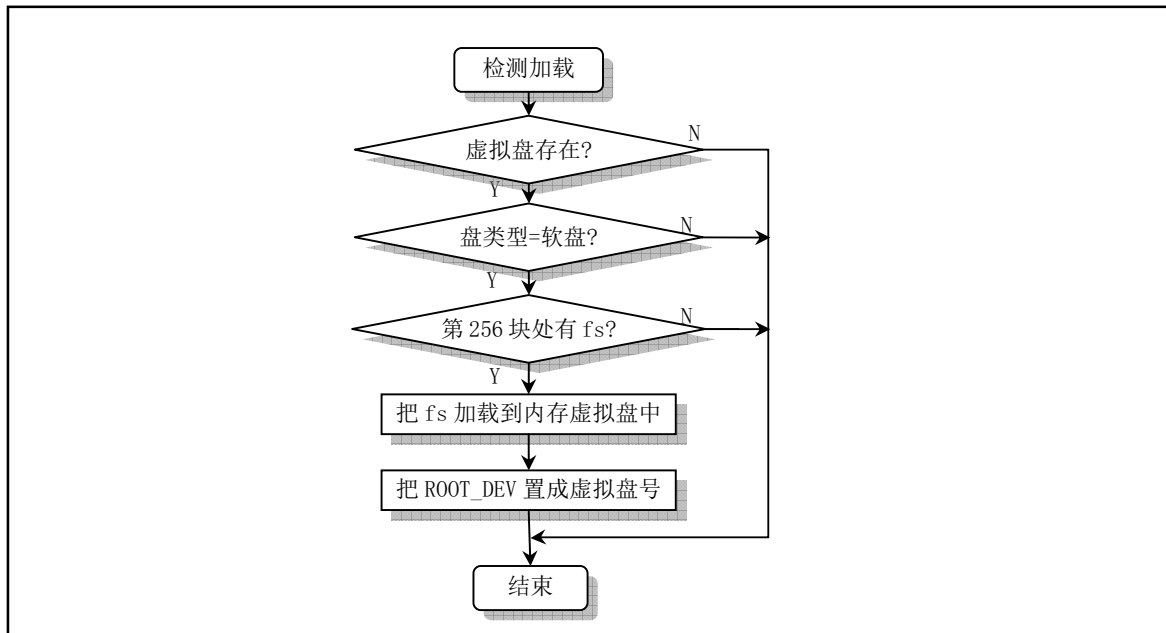


图 17-11 加载根文件系统到内存虚拟盘区域的流程图

17.10.1.2 集成盘的结构

对于 Linux 0.1x 内核，其代码加数据段的长度很小，大约在 120KB 左右。在开发 Linux 系统初始阶段，即使考虑到内核的扩展，Linus 还是认为内核的长度不会超过 256KB，因此在 1.44MB 的盘上可以放一个基本的根文件系统放在启动盘的第 256 个磁盘块开始的地方，组合形成一个集成盘片。一个添加了基本根文件系统的引导盘（即集成盘）的结构示意图见图 17-12 所示。其中文件系统的详细结构请参见文件系统一章中的说明。

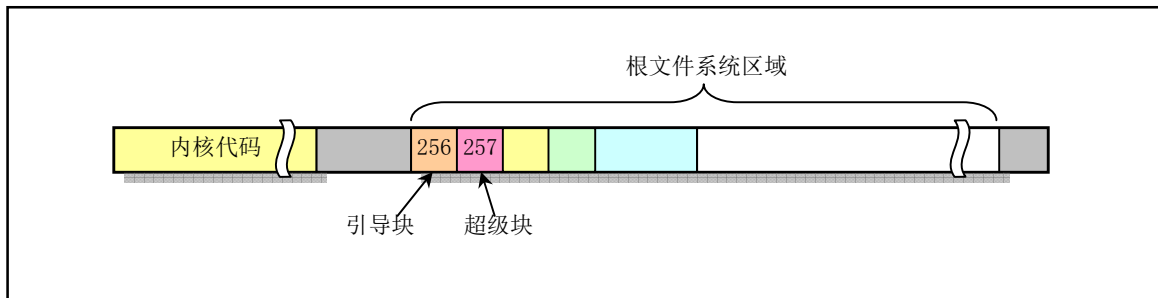


图 17-12 集成盘上代码结构

如上所述，集成盘上根文件系统放置的位置和大小主要与内核的长度和定义的 RAMDISK 区域的大小有关。Linux 在 ramdisk.c 程序中默认地定义了这个根文件的开始放置位置为第 256 磁盘块开始的地方。对于 Linux 0.11 内核来讲，编译产生的内核 Image 文件（即引导启动盘 Image 文件）的长度在 120KB 左右，因此把根文件系统放在盘的第 256 磁盘块开始的地方肯定没有问题，只是稍许浪费了一点磁盘空间。还剩下共有 $1440 - 256 = 1184$ KB 空间可用来存放根文件系统。当然我们也可以根据具体编译出的内核大小来调整存放根文件的开始磁盘块位置。例如我们可以修改 ramdisk.c 第 75 行 block 的值为 130 把存放根文件的开始位置往前挪动一些以腾出更多的磁盘空间供盘上的根文件系统使用。

17.10.2 集成盘的制作过程

在不改动内核程序 ramdisk.c 中默认定义的根文件系统开始存放磁盘块位置的情况下，我们假设需要制作集成盘上的根文件系统的容量为 1024KB（最大不超过 1184KB）。制作集成盘的主要思路是首先建立一个 1.44MB 的空的 Image 盘文件，然后将新编译出的开启了 RAMDISK 功能的内核 Image 文件复制到该盘的开始处。再把定制的大小不超过 1024KB 的文件系统复制到该盘的第 256 磁盘块开始处。具体制作步骤如下所示。

17.10.2.1 重新编译内核

重新编译带有 RAMDISK 定义的内核 Image 文件，假定 RAMDISK 区域设置为 2048KB。方法如下：在 Bochs 系统中运行 linux-0.11-devel 系统。编辑其中的 /usr/src/linux/Makefile 文件，修改以下设置行：

```
RAMDISK = -DRAMDISK = 2048
ROOT_DEV = FLOPPY
```

然后重新编译内核源代码生成新的内核 Image 文件。

```
make clean; make
```

17.10.2.2 制作临时根文件系统

制作大小为 1024KB 的根文件系统 Image 文件，假定其文件名为 rootram.img。在制作过程中使用带

硬盘 Image 文件的 Bochs 配置文件 (bochsrc-hd.bxrc) 运行 Bochs 系统。制作方法如下:

(1) 利用本章前面介绍的方法制作一张大小为 1024KB 的空 Image 文件。假定该文件的名称是 rootram.img。可使用在现在的 Linux 系统下执行下面命令生成:

```
dd bs=1024 if=/dev/zero of=rootram.img count=1024
```

(2) 在 Bochs 系统中运行 linux-0.11-devel 系统。然后在 Bochs 主窗口上把驱动盘分别配置成: A 盘为 rootimage-0.11-orign; B 盘为 rootram.img。

(3) 使用下面命令在 rootram.img 盘上创建大小为 1024KB 的空文件系统。然后分别把 A 盘和 B 盘加载到 /mnt 和 /mnt1 目录上。若目录 /mnt1 不存在, 可以建立一个。

```
mkfs /dev/fd1 1024
mkdir /mnt1
mount /dev/fd0 /mnt
mount /dev/fd1 /mnt1
```

(4) 使用 cp 命令有选择性地复制 /mnt 上 rootimage-0.11-orign 中的文件到 /mnt1 目录中, 在 /mnt1 中制作出一个根文件系统。若遇到出错信息, 那么通常是容量已经超过了 1024KB 了。利用下面的命令或使用本章前面介绍的方法来建立根文件系统。

首先精简 /mnt/ 中的文件, 以满足容量不要超过 1024KB 的要求。我们可以删除一些 /bin 和 /usr/bin 下的文件来达到这个要求。关于容量可以使用 df 命令来查看。例如我选择保留的文件是以下一些:

```
[/mnt/bin]# ll
total 495
-rwx--x--x  1 root    root      29700 Apr 29 20:15 mkfs
-rwx--x--x  1 root    root      21508 Apr 29 20:15 mknod
-rwx--x--x  1 root    root      25564 Apr 29 20:07 mount
-rwxr-xr-x  1 root    root     283652 Sep 28 10:11 sh
-rwx--x--x  1 root    root      25646 Apr 29 20:08 umount
-rwxr-xr-x  1 root    root      4096   116479 Mar  3  2004 vi
[/mnt/bin]# cd /mnt/usr/bin
[/mnt/usr/bin]# ll
total 364
-rwxr-xr-x  1 root    root      29700 Jan 15  1992 cat
-rwxr-xr-x  1 root    root      29700 Mar  4  2004 chmod
-rwxr-xr-x  1 root    root     33796 Mar  4  2004 chown
-rwxr-xr-x  1 root    root     37892 Mar  4  2004 cp
-rwxr-xr-x  1 root    root      29700 Mar  4  2004 dd
-rwx--x--x  1 root    root      36125 Mar  4  2004 df
-rwx--x--x  1 root    root     46084 Sep 28 10:39 ls
-rwxr-xr-x  1 root    root      29700 Jan 15  1992 mkdir
-rwxr-xr-x  1 root    root     33796 Jan 15  1992 mv
-rwxr-xr-x  1 root    root      29700 Jan 15  1992 rm
-rwxr-xr-x  1 root    root     25604 Jan 15  1992 rmdir
[/mnt/usr/bin]#
```

然后利用下列命令复制文件。另外, 可以按照自己的需要修改一下 /mnt/etc/fstab 和 /mnt/etc/rc 文件中的内容。此时, 我们就在 fd1 (/mnt1/) 中建立了一个大小在 1024KB 以内的文件系统。

```
cd /mnt1
for i in bin dev etc usr tmp
do
cp +recursive +verbose /mnt/$i $i
done
sync
```

(5) 使用 `umount` 命令卸载 `/dev/fd0` 和 `/dev/fd1` 上的文件系统, 然后使用 `dd` 命令把 `/dev/fd1` 中的文件系统复制到 Linux-0.11-devel 系统中, 建立一个名称为 `rootram-0.11` 的根文件系统 Image 文件:

```
dd bs=1024 if=/dev/fd1 of=rootram-0.11 count=1024
```

此时在 Linux-0.11-devel 系统中我们已经有了新编译出的内核 Image 文件 `/usr/src/linux/Image` 和一个简单的容量不超过 1024KB 的根文件系统映像文件 `rootram-0.11`。

17.10.2.3 建立集成盘

组合上述两个映像文件, 建立集成盘。修改 Bochs 主窗口 A 盘配置, 将其设置为前面准备好的 1.44MB 名称为 `bootroot-0.11` 的映像文件。然后执行命令:

```
dd bs=8192 if=/usr/src/linux/Image of=/dev/fd0
dd bs=1024 if=rootram-0.11 of=/dev/fd0 seek=256
sync;sync;sync;
```

其中选项 `bs=1024` 表示定义缓冲的大小为 1KB。`seek=256` 表示写输出文件时跳过前面的 256 个磁盘块。然后退出 Bochs 系统。此时我们在主机的当前目录下就得到了一张可以运行的集成盘映像文件 `bootroot-0.11`

17.10.3 运行集成盘系统

先为集成盘制作一个简单的 Bochs 配置文件 `bootroot-0.11.bxrc`。其中主要设置是:

```
floppya: 1_44=bootroot-0.11
```

然后用鼠标双击该配置文件运行 Bochs 系统。此时应有如图 17-13 所示显示结果。

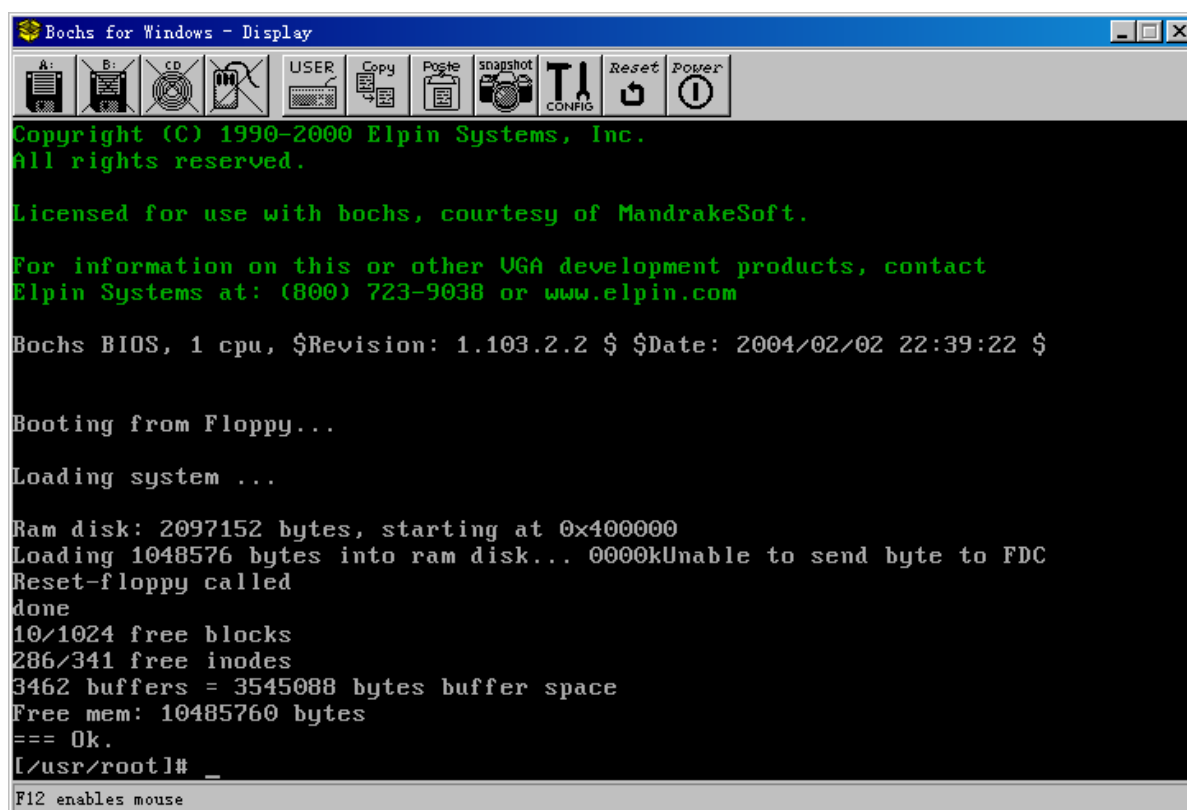


图 17-13 集成盘运行界面

为了方便大家做实验，也可以从下面网址下载已经做好并能立刻运行的集成盘软件：

<http://oldlinux.org/Linux.old/bochs/bootroot-0.11-040928.zip>

17.11 从硬盘启动：利用 shoelace 引导软件

很多人想从硬盘 Image 文件直接能引导启动 Linux 0.11 系统。本节介绍如何使用引导启动软件 shoelace 来制作这样的—个硬盘 Image 文件。shoelace 类似于 Linux 使用的 LILO 或 Grub 引导系统。它原来是 MINIX 系统的硬盘引导系统，后来于 1992 年 1 月份被移植到 Linux 上。有关硬盘引导程序 shoelace（或 grub, Lilo 等）实现原理请参考其它资料，例如 boot-HOWTO 等。shoelace.tar.z 可以从 Linux.old/bin-src/ 中下载。

17.11.1 shoelace 程序设置路径

shoelace 软件包主要包括以下几个程序，并需要按照以下路径放置在实验系统中：

```
/etc/laceup
/etc/bootlace
/etc/winiboot
/etc/config
/etc/disktab
```

```
/shoelace
```

除了/etc/config 文件需要修改,其他文件一般均不需要改动。只需修改 config 文件中 boot: 一行上的内容,将其指向系统 bootimage 文件所在的路径位置,即内核源代码编译出的 Image 文件,通常其路径名是 /usr/src/linux/Image

17.11.2 设置过程

假如我们需要从硬盘的第 1 个分区启动,那么执行以下操作就可以完成 shoelace 的执行:

```
/etc/laceup /dev/hd1 wini
```

这行命令将在硬盘第 1 个分区引导扇区中放置 shoelace 的引导扇区程序。如果需要修改硬盘的主引导扇区,让你可以选择使用哪个分区引导,那么你还需要执行:

```
/etc/laceup -w 1 /dev/hd0
```

其中"-w 1"表示默认选择第 1 个分区作为引导分区。

17.11.3 问题和解决方法

执行 fdisk 命令对硬盘 Image 文件进行分区操作时,如果你使用的是 MINIX 环境(例如 MINIX-1.5 系统),并且没有出现任何疑问提示,那么通过上述操作,这个 hd image 文件肯定能够被引导启动。如果使用的是 Linux 0.11 上的 fdisk,则可能会碰到问题。主要原因是 MINIX 的 fdisk 对分区操作有些特别。分区的参数需要进行一些特别选择才行。由于网站上现有的 Linux-0.11-devel-XXXX.zip(2004 年 9 月 23 日之前的版本)中的 hd image 文件参数和分区与 MINIX 做出的有些不同,因此使用上面操作后硬盘并不能正常启动。因此最好重新建立一个 hd 映像文件,并且该硬盘映像文件的参数经过一些慎重选择。然后在其上建立一个根文件系统。通过实验,我们可以使用以下参数从硬盘上正常引导:

硬盘 Image 文件参数: cyl = 410, heads=16, sectors=38。

这个硬盘大小在 127MB 左右。对硬盘 Image 文件进行分区操作之后,在 MINIX 系统下分区设置的显示值为:

```
[root /]# fdisk -h16 -s38 /dev/hd5
```

			----first----	----last----	-----sectors-----							
Num	Sorted	Active	Type	Cyl	Head	Sec	Cyl	Head	Sec	Base	Last	Size
1	1		MINIX	0	0	3	203	15	38	2	124031	124030
2	2		MINIX	204	0	1	407	15	38	124032	248063	124032
3	3		None	0	0	0	0	0	0	0	0	0
4	4		None	0	0	0	0	0	0	0	0	0

在 Linux 0.11 上执行 fdisk 命令的显示值为:

```
[/usr/root]# fdisk
p
Command (m for help):
Disk /dev/hd0: 0 heads, 0 sectors, 0 cylinders
```

Device	Boot	Begin	Start	End	Blocks	Id	System
/dev/hd01		0	2	124031	62015	81	Linux/MINIX
/dev/hd02		0	124032	248063	62016	81	Linux/MINIX

x

Command (m for help):

p

Expert command (m for help):

Disk /dev/hd0: 0 heads, 0 sectors, 0 cylinders

Nr	AF	Hd	Sec	Cyl	Hd	Sec	Cyl	Start	Size	ID
1	00	0	3	0	15	38	203	2	124030	81
2	00	0	1	204	15	38	407	124032	124032	81
3	00	0	0	0	0	0	0	0	0	00
4	00	0	0	0	0	0	0	0	0	00

你可以在 Linux 系统下先建立一个全零值 hd image 文件 hdc.img:

```
dd if=/dev/zero of=hdc.img bs=512 count=248280
```

上面 count 是扇区数($=410 * 16 * 38$)。然后按照上面参数建立分区。在把该硬盘 Image 文件 mount 到 Linux 0.11 系统后，直接 `cp -a` 完全复制/目录即可。例如，如果新硬盘 Image 文件是被加载到了/mnt/目录上，那么执行：

```
[/usr/root]# cd /mnt/
[/mnt]# cp -a /* .
```

最后可能会碰到的一个问题是需要给主引导扇区加上引导扇区标志 0x55,0xAA（第 511、512 字节）刻意使用 UltraEdit 编辑 hdc.img 加上。已经制作好的可启动硬盘 Image 文件包可从本书的网站下载：

<http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040923.zip>

启动显示样子见图 17-14 所示。

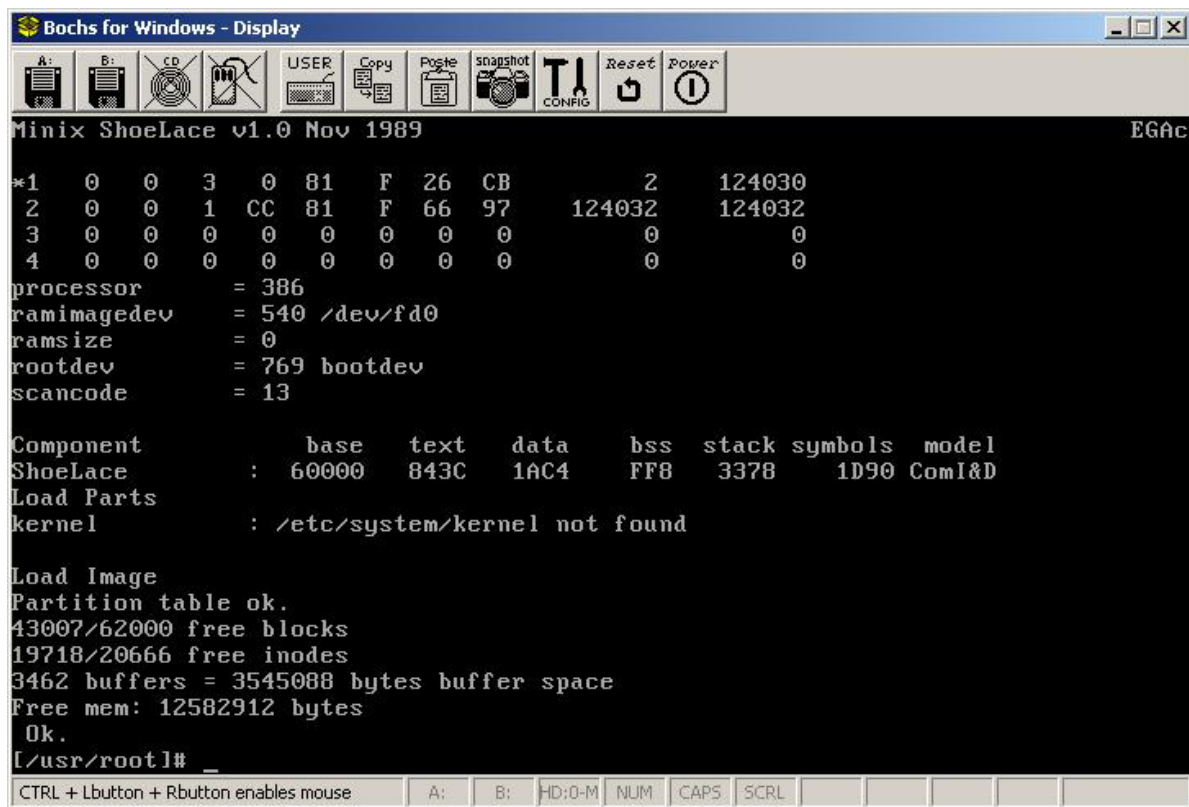


图 17-14 利用 shoelace 从硬盘引导启动 Linux 0.11 系统

17.12 利用 GDB 和 Bochs 调试内核源代码

本节说明如何在现有 Linux 系统(例如 RedHat 9)上使用 Bochs 模拟运行环境和 gdb 工具来调试 Linux 0.11 内核源代码。在使用这个方法之前,你的 Linux 系统上应该已经安装有 X window 系统。由于 Bochs 网站提供的 RPM 安装包中的 Bochs 执行程序没有编译进与 gdb 调试器进行通信的 gdbstub 模块,因此我们需要下载 Bochs 源代码来自行编译。

gdbstub 可以使得 Bochs 程序在本地 1234 网络端口侦听接收 gdb 的命令,并且向 gdb 发送命令执行结果。从而我们可以利用 gdb 对 Linux 0.11 内核进行 C 语言级的调试。当然, Linux 0.11 内核也需要进行使用 -g 选项重新编译。

17.12.1 编译带 gdbstub 的 Bochs 系统

Bochs 用户手册中介绍了自行编译 Bochs 系统的方法。这里我们给出编译带 gdbstub 的 Bochs 系统的方法和步骤。首先从下面网站下载最新 Bochs 系统源代码(例如: bochs-2.2.tar.gz):

<http://sourceforge.net/projects/bochs/>

使用 tar 对软件包解压后会在当前目录中生成一个 bochs-2.2 子目录。进入该子目录后带选项 “--enable-gdb-stub” 运行配置程序 configure, 然后运行 make 和 make install 即可, 见如下所示:

```
[root@plinux bochs-2.2]# ./configure --enable-gdb-stub
```

```
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
...
[root@plinux bochs-2.2]# make
[root@plinux bochs-2.2]# make install
```

若在运行 `./configure` 时我们碰到一些问题而不能生成编译使用的 `Makefile` 文件，那么这通常是由于没有安装 X window 开发环境软件或相关库文件造成的。此时我们就必须先安装这些必要的软件。

17.12.2 编译带调试信息的 Linux 0.11 内核

通过把 Bochs 的模拟运行环境与 gdb 符号调试工具联系起来，我们既可以使用 Linux 0.11 系统下编译的带调试信息的内核模块来调试，也可以使用在 RedHat 9 环境下编译的 0.11 内核模块来调试。这两种环境下都需要对 0.11 内核源代码目录中所有 `Makefile` 文件进行修改，即在其中编译标志行上添加 `-g` 标志，并去掉链接标志行上的 `-s` 选项：

```
LDFLAGS = -M -x                                // 去掉 -s 标志。
CFLAGS  =-Wall -O -g -fomit-frame-pointer \      // 添加 -g 标志。
```

进入内核源代码目录后，利用 `find` 命令我们可以找到以下所有需要修改的 `Makefile` 文件：

```
[root@plinux linux-0.11]# find ./ -name Makefile
./fs/Makefile
./kernel/Makefile
./kernel/chr_drv/Makefile
./kernel/math/Makefile
./kernel/blk_drv/Makefile
./lib/Makefile
./Makefile
./mm/Makefile
[root@plinux linux-0.11]#
```

另外，由于此时编译出的内核代码模块中含有调试信息，因此 `system` 模块大小可能会超过写入内核代码映像文件的默认最大值 `SYSSIZE = 0x3000`（定义在 `boot/bootsect.s` 文件第 6 行）。我们可以按以下方法修改源代码根目录中的 `Makefile` 文件中产生 `Image` 文件的规则，即把内核代码模块 `system` 中的符号信息去掉后再写入 `Image` 文件中，而原始带符号信息的 `system` 模块保留用作 `gdb` 调试器使用。注意，目标的实现命令需要以一个制表符（`TAB`）作为一行的开始。

```
Image: boot/bootsect boot/setup tools/system tools/build
      cp -f tools/system system.tmp
      strip system.tmp
      tools/build boot/bootsect boot/setup system.tmp $(ROOT_DEV) $(SWAP_DEV) > Image
      rm -f system.tmp
      sync
```

当然，我们也可以把 `boot/bootsect.s` 和 `tools/build.c` 中的 `SYSSIZE` 值修改成 `0x8000` 来处理这种情况。

17.12.3 调试方法和步骤

下面我们根据在现代 Linux 系统（例如 RedHat 9）系统上和运行在 Bochs 中 Linux 0.11 系统上编译出的内核代码分别来说明调试方法和步骤。

17.12.3.1 调试现代 Linux 系统上编译出的 Linux 0.11 内核

假设我们的 Linux 0.11 内核源代码根目录是 linux-rh9-gdb/, 则我们首先在该目录中按照上面方法修改所有 Makefile 文件, 然后在 linux-rh9-gdb/目录下创建一个 bochs 运行配置文件并下载一个配套使用的根文件系统映像文件。我们可以直接从网站下载已经设置好的如下软件包来做实验:

<http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-rh9-050619.tar.gz>

使用命令 “tar zxvf linux-gdb-rh9-050619.tar.gz” 解开这个软件包后, 可以看到其中包含以下几个文件和目录:

```
[root@plinux linux-gdb-rh9]# ll
total 1600
-rw-r--r-- 1 root root 18055 Jun 18 15:07 bochsrc-fdl-gdb.bxrc
drwxr-xr-x 10 root root 4096 Jun 18 22:55 linux
-rw-r--r-- 1 root root 1474560 Jun 18 20:21 rootimage-0.11-for-orig
-rwxr-xr-x 1 root root 35 Jun 18 16:54 run
[root@plinux linux-gdb-rh9]#
```

第 1 个文件 bochsrc-fdl-gdb.bxrc 是 Bochs 配置文件, 其中已经把文件系统映像文件 rootimage-0.11-for-orig 设置为插入在第 2 个“软盘驱动器”中。这个 bochs 配置文件与其他 Linux 0.11 配置文件的主要区别是在文件头部添加有以下一行内容, 表示当 bochs 使用这个配置文件运行时将在本地网络端口 1234 上侦听 gdb 调试器的命令:

```
gdbstub: enabled=1, port=1234, text_base=0, data_base=0, bss_base=0
```

上面第 2 项 linux/是 Linux 0.11 源代码目录, 其中包含了已经修改好所有 Makefile 文件的内核源代码文件。第 3 个文件 rootimage-0.11-for-orig 是与这个内核代码配套的根文件系统映像文件。第 4 个文件是一个简单脚本程序, 其中包含一行 bochs 启动命令行。运行这个实验的基本步骤如下:

1. 启动 X window 系统后打开两个终端窗口;
2. 在一个窗口中, 把工作目录切换进 linux-gdb-rh9/目录中, 并运行程序 “./run”, 此时该窗口中会显示一条等待 gdb 来连接的信息: “Wait for gdb connection on localhost:1234”, 并且系统会创建一个 Bochs 主窗口 (此时无内容);
3. 在另一个窗口中, 我们把工作目录切换到内核源代码目录中 linux-gdb-rh9/linux/, 并运行命令: “gdb tools/system”;
4. 在运行 gdb 的窗口中键入命令 “break main” 和 “target remote localhost:1234”, 此时 gdb 会显示已经连接到 Bochs 的信息;
5. 在 gdb 环境中再执行命令 “cont”, 稍过一会 gdb 会显示程序停止在 init/main.c 的 main()函数处。

此后我们就可以使用 gdb 的命令来观察源代码和调试内核程序了。例如我们可以使用 list 命令来观察源代码、用 help 命令来取得在线帮助信息、用 break 来设置其他断点、用 print/set 来显示/设置一些变量值、用 next/step 来执行单步调试、用 quit 命令退出 gdb 等。gdb 的具体使用方法请参考 gdb 手册。下面是运行 gdb 和在其中执行的一些命令示例。

```

[root@plinux linux]# gdb tools/system // 启动 gdb 执行 system 内核模块。
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) break main // 在 main() 函数处设置断点。
Breakpoint 1 at 0x6621: file init/main.c, line 110.
(gdb) target remote localhost:1234 // 与 Bochs 连接。
Remote debugging using localhost:1234
0x0000fff0 in sys_mkdir (pathname=0x0, mode=0) at namei.c:481
481 namei.c: No such file or directory.
    in namei.c
(gdb) cont // 继续执行至断点处。
Continuing.
Breakpoint 1, main () at init/main.c:110 // 程序在断点处停止运行。
110 ROOT_DEV = ORIG_ROOT_DEV;
(gdb) list // 查看源代码。
105 { /* The startup routine assumes (well, ...) this */
106 /*
107  * Interrupts are still disabled. Do necessary setups, then
108  * enable them
109  */
110     ROOT_DEV = ORIG_ROOT_DEV;
111     drive_info = DRIVE_INFO;
112     memory_end = (1<<20) + (EXT_MEM_K<<10);
113     memory_end &= 0xfffff000;
114     if (memory_end > 16*1024*1024)
(gdb) next // 单步执行。
111     drive_info = DRIVE_INFO;
(gdb) next // 单步执行。
112     memory_end = (1<<20) + (EXT_MEM_K<<10);
(gdb) print /x ROOT_DEV // 打印变量 ROOT_DEV 的值。
$3 = 0x21d // 第 2 个软盘设备号。
(gdb) quit // 退出 gdb 命令。
The program is running. Exit anyway? (y or n) y
[root@plinux linux]#

```

在 `gdb` 中进行内核源代码调试时，有时会显示源程序没有找到的问题，例如 `gdb` 有时会显示“`memory.c: No such file or directory`”，这是由于在编译 `mm/` 下的 `memory.c` 等文件时，`Makefile` 文件指示 `ld` 链接器把生成的 `mm/` 下的文件模块链接生成了可重定位的模块 `mm.o`，并且在源代码根目录 `linux/` 下再次作为 `ld` 的输入模块。因此，我们可以把这些文件复制到 `linux/` 目录下重新进行调试操作。

17.12.3.2 调试 Linux 0.11 系统上编译出的 0.11 内核

为了在 RedHat 9 等现代 Linux 操作系统中调试 Linux 0.11 系统上编译出的内核，那么我们在修改和编译出内核映像文件 `Image` 之后，需要把整个 0.11 内核源代码目录复制到 Redhat 9 系统中。然后按照上面类似的步骤进行操作。我们可以使用前面介绍的 `linux-0.11-devel` 环境编译内核，然后对这个包含 `Image` 文件的内核源代码目录树进行压缩，再使用 `mcoppy` 命令写到 Bochs 的第 2 个软盘映像文件中，最后利用

WinImage 软件或者 mount 命令取出其中的压缩文件。下面给出编译和取出文件过程的基本步骤。

1. 在 Bochs 下运行 Linux-0.11-devel 系统，进入目录/usr/src/，并创建目录 linux-gdb；
2. 使用命令先复制整个 0.11 内核源代码树：“cp -a linux linux-gdb/”。然后进入 linux-gdb/linux/目录，按照上面所述方法修改所有 Makefile 文件，并编译内核；
3. 回到/usr/src/目录，用 tar 命令对 linux-gdb/目录进行压缩，得到 linux-gdb.tgz 文件；
4. 把压缩文件复制到第 2 个软盘 (b 盘)映像文件中：“mcopy linux-gdb.tgz b:”。如果 b 盘空间不够，请先使用删除文件命令 “mdel b: 文件名”在 b 盘上腾出一些空间。
5. 如果主机环境是 windows 操作系统，那么请使用 WinImage 取出 b 盘映像文件中的压缩文件，并通过 FTP 服务器或使用其他方法放到 Redhat 9 系统中；如果主机环境原来就是 Redhat 9 或其他现代 Linux 系统，那么可以使用 mount 命令加载 b 盘映像文件，并从中复制出压缩内核文件。
6. 在现代 Linux 系统上解压复制出的压缩文件，会生成包含 0.11 内核源代码目录树的 linux-gdb/目录。进入 linux-gdb/目录，创建 bochs 配置文件 bochsrc-fd1-gdb.bxrc。这个配置文件也可以直接取自于 linux-0.11-devel 软件包中的 bochsrc-fdb.bxrc 文件，并自行添加上 gdbstub 参数行。再从 oldlinux.org 网站上下载 rootimage-0.11 根文件系统软盘映像文件，同样保存在 linux-gdb/目录中。

此后我们可以继续按照上一小节的步骤执行源代码调试实验。下面是上述步骤的一个示例，我们假设主机环境是 Redhat 9 系统，并在其上运行 Bochs 中的 Linux 0.11 系统。

```

[/usr/root]# cd /usr/src                // 进入源代码目录。
[/usr/src]# mkdir linux-gdb             // 创建目录 linux-gdb。
[/usr/src]# cp -a linux linux-gdb/      // 把内核源代码树复制到 linux-gdb/中。
[/usr/src]# cd linux-gdb/linux
[/usr/src/linux-gdb/linux]# vi Makefile // 修改所有 Makefile 文件。
...
[/usr/src/linux-gdb/linux]# make clean; make // 编译内核。
...
[/usr/src/linux-gdb/linux]# cd ../../
[/usr/src]# tar zcvf linux-gdb.tgz linux-gdb // 创建压缩文件 linux-gdb.tgz。
...
[/usr/src]# mdir b:                     // 查看 b 盘映像文件内容。
Volume in drive B is Bt
Directory for B:/
LINUX-GD TGZ      827000    6-18-105   10:28p
TPUT      TAR      184320    3-09-132    3:16p
LILO      TAR      235520    3-09-132    6:00p
SHOELA~1 Z       101767    9-19-104    1:24p
SYSTEM     MAP       17771   10-05-104   11:22p
      5 File(s)      90624 bytes free
[/usr/src]# mdel b:linux-gd.tgz         // 空间不够，于是删除 b 盘上文件。
[/usr/src]# mcopy linux-gdb.tgz b:      // 把 linux-gdb.tgz 复制到 b 盘上。
Copying LINUX-GD.TGZ
[/usr/src]#

```

关闭 Bochs 系统后，我们在 b 盘映像文件中得到名称为 LINUX-GD.TGZ 的压缩文件。在 Redhat 9 Linux 主机环境下利用下面命令序列可以建立起调试实验目录。

```

[root@plinux 0.11]# mount -t msdos diskb.img /mnt/d4 -o loop,r // 加载 b 盘映像文件。
[root@plinux 0.11]# ls -l /mnt/d4                               // 查看 b 盘中内容。

```

```

total 1234
-rwxr-xr-x    1 root    root      235520 Mar  9  2032 lilo.tar
-rwxr-xr-x    1 root    root      723438 Jun 19  2005 linux-gd.tgz
-rwxr-xr-x    1 root    root     101767 Sep 19  2004 shoela~1.z
-rwxr-xr-x    1 root    root      17771 Oct  5  2004 system.map
-rwxr-xr-x    1 root    root     184320 Mar  9  2032 tput.tar
[root@plinux 0.11]# cp /mnt/d4/linux-gd.tgz .           // 复制 b 盘中的压缩文件。
[root@plinux 0.11]# umount /mnt/d4                     // 卸载 b 盘映像文件。
[root@plinux 0.11]# tar zxvf linux-gd.tgz              // 解压文件。
...
[root@plinux 0.11]# cd linux-gdb
[root@plinux linux-gdb]# ls -l
total 4
drwx--x--x   10 15806    root        4096 Jun 19  2005 linux
[root@plinux linux-gdb]#

```

此后我们还需要在 linux-gdb/目录下创建 Bochs 运行配置文件 bochsrc-fd1-gdb.bxrc, 并且下载软盘根文件系统映像文件 rootimage-0.11。为方便起见, 我们还可以创建只包含“bochs -q -f bochsrc-fd1-gdb.bxrc”一行内容的脚本文件 run, 并把该文件属性设置成可执行。另外, oldlinux.org 上已经为大家制作好一个可以直接进行调试实验的软件包, 其中包含的内容与直接在 Redhat 9 下编译使用的软件包内容基本相同:

<http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-050619.tar.gz>

参考文献

- [1] Intel Co. INTEL 80386 Programmer's Reference Manual 1986, INTEL CORPORATION,1987.
- [2] Intel Co. IA-32 Intel Architecture Software Developer's Manual Volume.3: System Programming Guide. <http://www.intel.com/>, 2005.
- [3] James L. Turley. Advanced 80386 Programming Techniques. Osborne McGraw-Hill,1988.
- [4] Brian W. Kernighan, Dennis M. Ritchie. The C programming Language. Prentice-Hall 1988.
- [5] Leland L. Beck. System Software: An Introduction to Systems Programming,3nd. Addison-Wesley,1997.
- [6] Richard Stallman, Using and Porting the GNU Compiler Collection,the Free Software Foundation, 1998.
- [7] The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001, The IEEE and The Open Group.
- [8] David A Rusling, The Linux Kernel, 1999. <http://www.tldp.org/>
- [9] Linux Kernel Source Code, <http://www.kernel.org/>
- [10] Digital co.ltd. VT100 User Guide, <http://www.vt100.net/>
- [11] Clark L. Coleman. Using Inline Assembly with gcc. <http://oldlinux.org/Linux.old/>
- [12] John H. Crawford, Patrick P. Gelsinger. Programming the 80386. Sybex, 1988.
- [13] FreeBSD Online Manual, <http://www.freebsd.org/cgi/man.cgi>
- [14] Andrew S.Tanenbaum 著 陆佑珊、施振川译, 操作系统教程 MINIX 设计与实现. 世界图书出版公司, 1990.4
- [15] Maurice J. Bach 著, 陈葆珏, 王旭, 柳纯录, 冯雪山译, UNIX 操作系统设计. 机械工业出版社, 2000.4
- [16] John Lions 著, 尤晋元译, 莱昂氏 UNIX 源代码分析, 机械工业出版社, 2000.7
- [17] Andrew S. Tanenbaum 著 王鹏, 尤晋元等译, 操作系统: 设计与实现 (第 2 版), 电子工业出版社, 1998.8
- [18] Alessandro Rubini, Jonathan 著, 魏永明, 骆刚, 姜君译, Linux 设备驱动程序, 中国电力出版社, 2002.11
- [19] Daniel P. Bovet, Marco Cesati 著, 陈莉君, 冯锐, 牛欣源 译, 深入理解 LINUX 内核, 中国电力出版社 2001.
- [20] 张载鸿. 微型机(PC 系列)接口控制教程, 清华大学出版社, 1992.
- [21] 李凤华, 周利华, 赵丽松. MS-DOS 5.0 内核剖析. 西安电子科技大学出版社, 1992.
- [22] RedHat 9.0 操作系统在线手册. <http://www.plinux.org/cgi-bin/man.cgi>
- [23] W.Richard Stevens 著 尤晋元等译, UNIX 环境高级编程. 机械工业出版社, 2000.2
- [24] Linux Weekly Edition News. <http://lwn.net/>
- [25] P.J. Plauger. The Standard C Library. Prentice Hall, 1992
- [26] Free Software Foundation. The GNU C Library. <http://www.gnu.org/> 2001
- [27] Chuck Allison. The Standard C Library. C/C++ Users Journal CD-ROM, Release 6. 2003
- [28] Bochs simulation system. <http://bochs.sourceforge.net/>
- [29] Brennan "Bas" Underwood. Brennan's Guide to Inline Assembly. <http://www.rt66.com/~brennan/>
- [30] John R. Levine. Linkers & Loaders. <http://www.iecc.com/linker/>
- [31] Randal E. Bryant, David R. O'Hallaron 著. 龚奕利, 雷迎春译. 深入理解计算机系统. 中国电力出版社 2004
- [32] Randal E. Bryant, David R. O'Hallaron. Computer Systems A programmer's Perspective. 电子工业出版社. 2004.
- [33] Intel. Data Sheet: 8254 Programmable Interval Timer. 1993.9
- [34] Intel. Data Sheet: 8259A Programmable Interrupt Controller. 1988.12
- [35] Intel. Data Sheet: 82077A CHMOS Single-chip Floppy Disk Controller. 1994.5
- [36] Robert Love 著, 陈莉君, 康华, 张波译, Linux 内核设计与实现, 机械工业出版社, 2004.11
- [37] Adam Chapweske. The PS/2 Keyboard Interface. <http://www.computer-engineering.org/>
- [38] Dean Elsner, Jay Fenlason & friends. Using as: The GNU Assembler. <http://www.gnu.org/> 1998

- [39] Steve Chamberlain. Using ld: The GNU linker. <http://www.gnu.org/> 1998
- [40] Michael K. Johnson. The Linux Kernel Hackers' Guide. <http://www.tldp.org/> 1995
- [41]

附录

附录1 内核数据结构

这里集中列出了内核中的主要数据结构，并给予简单说明，注明了每个结构所在的文件和具体位置。作为阅读时参考。

1. 执行文件结构 `a.out` (`include/a.out.h`, 第 6 行)

`a.out` (Assembly out) 执行文件头格式结构。

```
struct exec {
    unsigned long a_magic;        // 执行文件魔数。使用 N_MAGIC 等宏访问。
    unsigned a_text;             // 代码长度，字节数。
    unsigned a_data;             // 数据长度，字节数。
    unsigned a_bss;              // 文件中的未初始化数据区长度，字节数。
    unsigned a_syms;             // 文件中的符号表长度，字节数。
    unsigned a_entry;            // 执行开始地址。
    unsigned a_trsize;           // 代码重定位信息长度，字节数。
    unsigned a_drsize;           // 数据重定位信息长度，字节数。
};
```

2. 文件锁定操作结构 `flock` (`include/fcntl.h`, 43 行)

文件锁定操作数据结构。

```
struct flock {
    short l_type;                // 锁定类型 (F_RDLCK, F_WRLCK, F_UNLCK)。
    short l_whence;              // 开始偏移 (SEEK_SET, SEEK_CUR 或 SEEK_END)。
    off\_t l_start;                // 阻塞锁定的开始处。相对偏移 (字节数)。
    off\_t l_len;                  // 阻塞锁定的大小；如果是 0 则为到文件末尾。
    pid\_t l_pid;                  // 加锁的进程 id。
};
```

3. `sigaction` 的数据结构 (`include/signal.h`, 48 行)

`sigaction` 的数据结构。

```
struct sigaction {
    void (*sa_handler)(int);
    sigset\_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

`sa_handler` 是对应某信号指定要采取的行动。可以是上面的 `SIG_DFL`，或者是 `SIG_IGN` 来忽略该信号，也可以是指向处理该信号函数的一个指针。

`sa_mask` 给出了对信号的屏蔽码，在信号程序执行时将阻塞对这些信号的处理。另外，引起触发信号处理的信号也将被阻塞，除非使用了 `SA_NOMASK` 标志。

`sa_flags` 指定改变信号处理过程的信号集。

`sa_restorer` 恢复函数指针，由函数库 `Libc` 提供，用于清理用户态堆栈。

4. 终端窗口大小属性结构 (include/termios.h, 36 行)

窗口大小(Window size)属性结构。在窗口环境中可用于基于屏幕的应用程序。ioctl 中的 TIOCGWINSZ 和 TIOCSWINSZ 可用来读取或设置这些信息。

```
struct winsize {
    unsigned short ws_row;        // 窗口字符行数。
    unsigned short ws_col;        // 窗口字符列数。
    unsigned short ws_xpixel;     // 窗口宽度，像素值。
    unsigned short ws_ypixel;     // 窗口高度，像素值。
};
```

5. termio(s)结构 (include/termios.h, 44 行)

AT&T 系统 V 的 termio 结构。其中控制字符数据长度 NCC = 8。

```
struct termio {
    unsigned short c_iflag;        // 输入模式标志。
    unsigned short c_oflag;        // 输出模式标志。
    unsigned short c_cflag;        // 控制模式标志。
    unsigned short c_lflag;        // 本地模式标志。
    unsigned char c_line;          // 线路规程（速率）。
    unsigned char c_cc[NCC];       // 控制字符数组。
};
```

POSIX 的 termios 结构（第 54 行）。其中控制字符数据长度 NCC = 17。

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

以上定义的两个终端数据结构 termio 和 termios 是分别属于两类 UNIX 系列(或克隆), termio 是在 AT&T 系统 V 中定义的, 而 termios 是 POSIX 标准指定的。两个结构基本一样, 只是 termio 使用短整数类型定义模式标志集, 而 termios 使用长整数定义模式标志集。由于目前这两种结构都在使用, 因此为了兼容性, 大多数系统都同时支持它们。另外, 以前使用的是一类似的 sgtty 结构, 目前已基本不用。

6. 时间结构 (include/time.h, 第 18 行)

```
struct tm {
    int tm_sec;                   // 秒数 [0, 59]。
    int tm_min;                   // 分钟数 [0, 59]。
    int tm_hour;                  // 小时数 [0, 59]。
    int tm_mday;                  // 1 个月的天数 [0, 31]。
    int tm_mon;                   // 1 年中月份 [0, 11]。
    int tm_year;                  // 从 1900 年开始的年数。
    int tm_wday;                  // 1 星期中的某天 [0, 6] (星期天 =0)。
    int tm_yday;                  // 1 年中的某天 [0, 365]。
    int tm_isdst;                 // 夏令时标志。
};
```

7. 文件访问/修改结构 (include/utime.h, 第 6 行)

```
struct utimbuf {
    time_t actime;           // 文件访问时间。从 1970.1.1:0:0:0 开始的秒数。
    time_t modtime;          // 文件修改时间。从 1970.1.1:0:0:0 开始的秒数。
};
```

8. 缓冲区头结构 buffer_head (include/linux/fs.h, 第 68 行)

缓冲区头数据结构。在程序中常用 bh 来表示 buffer_head 类型变量的缩写。

```
struct buffer_head {
    char * b_data;           // 指向数据块的指针（数据块为 1024 字节）。
    unsigned long b_blocknr;  // 块号。
    unsigned short b_dev;     // 数据源的设备号（0 表示未用）。
    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，1-已修改。
    unsigned char b_count;    // 使用该数据块的用户数。
    unsigned char b_lock;     // 缓冲区是否被锁定，0-未锁；1-已锁定。
    struct task_struct * b_wait; // 指向等待该缓冲区解锁的任务。
    struct buffer_head * b_prev; // 前一块（这四个指针用于缓冲区的管理）。
    struct buffer_head * b_next; // 下一块。
    struct buffer_head * b_prev_free; // 前一空闲块。
    struct buffer_head * b_next_free; // 下一空闲块。
};
```

9. 内存中磁盘索引节点结构 (include/linux/fs.h, 第 93 行)

这是在内存中的 i 节点结构。磁盘上的索引节点结构 d_inode 只包括前 7 项。

```
struct m_inode {
    unsigned short i_mode;    // 文件类型和属性(rwx 位)。
    unsigned short i_uid;     // 用户 id (文件拥有者标识符)。
    unsigned long i_size;     // 文件大小 (字节数)。
    unsigned long i_mtime;    // 文件修改时间 (自 1970.1.1:0 算起, 秒)。
    unsigned char i_gid;      // 组 id (文件拥有者所在的组)。
    unsigned char i_nlinks;    // 文件目录项链接数。
    unsigned short i_zone[9]; // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
                                // zone 是区的意思, 可译成区段, 或逻辑块。

    // 以下字段在内存中。
    struct task_struct * i_wait; // 等待该 i 节点的进程。
    unsigned long i_atime;       // 最后访问时间。
    unsigned long i_ctime;       // i 节点自身修改时间。
    unsigned short i_dev;        // i 节点所在的设备号。
    unsigned short i_num;        // i 节点号。
    unsigned short i_count;      // i 节点被使用的次数, 0 表示该 i 节点空闲。
    unsigned char i_lock;        // 锁定标志。
    unsigned char i_dirt;        // 已修改(脏)标志。
    unsigned char i_pipe;        // 管道标志。
    unsigned char i_mount;       // 安装标志。
    unsigned char i_seek;        // 搜寻标志(lseek 时)。
    unsigned char i_update;      // 更新标志。
};
```

10. 文件结构 (include/linux/fs.h, 第 116 行)

文件结构，用于在文件句柄与 i 节点之间建立关系。

```

struct file {
    unsigned short f_mode;           // 文件操作模式 (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 (文件描述符) 数。
    struct m_inode * f_inode;       // 指向对应 i 节点。
    off_t f_pos;                    // 文件位置 (读写偏移值)。
};

```

11. 磁盘超级块结构 (include/linux/fs.h, 第 124 行)

内存中磁盘超级块结构。磁盘上的超级块结构 d_super_block 只包括前 8 项。

```

struct super_block {
    unsigned short s_ninodes;       // 节点数。
    unsigned short s_nzones;       // 逻辑块数。
    unsigned short s_imap_blocks;   // i 节点位图所占用的数据块数。
    unsigned short s_zmap_blocks;   // 逻辑块位图所占用的数据块数。
    unsigned short s_firstdatazone; // 第一个数据逻辑块号。
    unsigned short s_log_zone_size; // log(数据块数/逻辑块)。(以 2 为底)。
    unsigned long s_max_size;       // 文件最大长度。
    unsigned short s_magic;         // 文件系统魔数。
// 以下字段仅在内存中。
    struct buffer_head * s_imap[8]; // i 节点位图缓冲块指针数组 (占用 8 块, 可表示 64M)。
    struct buffer_head * s_zmap[8]; // 逻辑块位图缓冲块指针数组 (占用 8 块)。
    unsigned short s_dev;           // 超级块所在的设备号。
    struct m_inode * s_isup;        // 被安装的文件系统根目录的 i 节点。(isup=super i)
    struct m_inode * s_imount;     // 被安装到的 i 节点。
    unsigned long s_time;          // 修改时间。
    struct task_struct * s_wait;    // 等待该超级块的进程。
    unsigned char s_lock;          // 被锁定标志。
    unsigned char s_rd_only;       // 只读标志。
    unsigned char s_dirt;          // 已修改(脏)标志。
};

```

12. 目录项结构 (include/linux/fs.h, 第 157 行)

文件目录项结构。

```

struct dir_entry {
    unsigned short inode;           // i 节点。
    char name[NAME_LEN];           // 文件名。
};

```

13. 硬盘分区表结构 (include/linux/hdreg.h, 第 52 行)

硬盘分区表结构。参见下面列表后信息。

```

struct partition {
    unsigned char boot_ind;         // 引导标志。0x80-该分区可引导操作系统。
    unsigned char head;            // 分区起始磁头号。
    unsigned char sector;          // 分区起始扇区号 (位 0-5) 和起始柱面号高 2 位 (位 6-7)。
    unsigned char cyl;             // 分区起始柱面号低 8 位。
};

```

```

unsigned char sys_ind;      // 分区类型字节。0x0b-DOS; 0x80-Old Minix; 0x83-Linux
unsigned char end_head;    // 分区的结束磁头号。
unsigned char end_sector;  // 结束扇区号(位 0-5)和结束柱面号高 2 位(位 6-7)。
unsigned char end_cyl;     // 结束柱面号低 8 位。
unsigned int start_sect;   // 分区起始物理扇区号(从 0 开始计)。
unsigned int nr_sects;     // 分区占用的扇区数。
};

```

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1--4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小和起止的柱面号、磁道号、扇区号和引导标志。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE--0x1FD 处。4 个分区中同时只能有一个分区是可引导的。

14. 段描述符结构 (include/linux/head.h, 第 4 行)

CPU 中描述符的简单格式。

```

struct desc_struct {          // 定义了段描述符的数据结构。该结构仅说明每个描述
    unsigned long a,b;       // 符是由 8 个字节构成，每个描述符表共有 256 项。
} desc_table[256];

```

15. i387 使用的结构 (include/linux/sched.h, 第 40 行)

这是数学协处理器使用的结构，主要用于保存进程切换时 i387 的执行状态信息。

```

struct i387_struct {
    long    cwd;              // 控制字(Control word)。
    long    swd;              // 状态字(Status word)。
    long    twd;              // 标记字(Tag word)。
    long    fip;              // 协处理器代码指针。
    long    fcs;              // 协处理器代码段寄存器。
    long    foo;              // 内存操作数的偏移位置。
    long    fos;              // 内存操作数的段值。
    long    st_space[20];     // 8 个 10 字节的协处理器累加器。
};

```

16. 任务状态段结构 (include/linux/sched.h, 第 51 行)

任务状态段数据结构 (参见附录)。

```

struct tss_struct {
    long    back_link;        /* 16 high bits zero */
    long    esp0;
    long    ss0;              /* 16 high bits zero */
    long    esp1;
    long    ss1;              /* 16 high bits zero */
    long    esp2;
    long    ss2;              /* 16 high bits zero */
    long    cr3;
    long    eip;
    long    eflags;
    long    eax, ecx, edx, ebx;
    long    esp;
    long    ebp;
    long    esi;
    long    edi;
};

```

```

long    es;                /* 16 high bits zero */
long    cs;                /* 16 high bits zero */
long    ss;                /* 16 high bits zero */
long    ds;                /* 16 high bits zero */
long    fs;                /* 16 high bits zero */
long    gs;                /* 16 high bits zero */
long    ldt;               /* 16 high bits zero */
long    trace_bitmap;      /* bits: trace 0, bitmap 16-31 */
struct i387\_struct i387;
};

```

17. 进程（任务）数据结构 task (include/linux/sched.h, 第 78 行)

这是任务（进程）数据结构，或称为进程描述符。

```

struct task_struct {
    long state;                //任务的运行状态 (-1 不可运行, 0 可运行(就绪), >0 已停止)。
    long counter;              // 任务运行时间计数(递减)(滴答数), 运行时间片。
    long priority;             // 运行优先数。任务开始运行时 counter=priority, 越大运行越长。
    long signal;               // 信号。是位图, 每个比特位代表一种信号, 信号值=位偏移值+1。
    struct sigaction sigaction[32]; // 信号执行属性结构, 对应信号将要执行的操作和标志信息。
    long blocked;              // 进程信号屏蔽码(对应信号位图)。
    int exit_code;              // 任务执行停止的退出码, 其父进程会取。
    unsigned long start_code;   // 代码段地址。
    unsigned long end_code;     // 代码长度(字节数)。
    unsigned long end_data;     // 代码长度 + 数据长度(字节数)。
    unsigned long brk;          // 总长度(字节数)。
    unsigned long start_stack;  // 堆栈段地址。
    long pid;                  // 进程标识号(进程号)。
    long father;               // 父进程号。
    long pgrp;                 // 进程组号。
    long session;              // 会话号。
    long leader;               // 会话首领。
    unsigned short uid;         // 用户标识号(用户 id)。
    unsigned short euid;        // 有效用户 id。
    unsigned short suid;        // 保存的用户 id。
    unsigned short gid;         // 组标识号(组 id)。
    unsigned short egid;        // 有效组 id。
    unsigned short sgid;        // 保存的组 id。
    long alarm;                 // 报警定时值(滴答数)。
    long utime;                 // 用户态运行时间(滴答数)。
    long stime;                 // 系统态运行时间(滴答数)。
    long cutime;                // 子进程用户态运行时间。
    long cstime;                // 子进程系统态运行时间。
    long start_time;            // 进程开始运行时刻。
    unsigned short used_math;    // 标志: 是否使用了协处理器。
    int tty;                    // 进程使用 tty 的子设备号。-1 表示没有使用。
    unsigned short umask;        // 文件创建属性屏蔽位。
    struct m_inode * pwd;        // 当前工作目录 i 节点结构。
    struct m_inode * root;       // 根目录 i 节点结构。
    struct m_inode * executable; // 执行文件 i 节点结构。
    unsigned long close_on_exec; // 执行时关闭文件句柄位图标志。(参见 include/fcntl.h)
    struct file * filp[NR_OPEN]; // 文件结构指针表, 最多 32 项。表项号即是文件描述符的值。
    struct desc_struct ldt[3];    // 任务局部描述符表。0-空, 1-代码段 cs, 2-数据和堆栈段 ds&ss。
};

```

```
    struct tss_struct tss;          // 进程的任务状态段信息结构。
};
```

18. tty 等待队列结构 (include/linux/tty.h, 第 16 行)

tty 等待队列数据结构。

```
struct tty_queue {
    unsigned long data;              // 等待队列缓冲区中当前数据指针（字符数[??]）。
                                    // 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; // 等待进程列表。
    char buf[TTY_BUF_SIZE];         // 队列的缓冲区。
};
```

19. tty 结构 (include/linux/tty.h, 第 45 行)

tty 数据结构。

```
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),
};
extern struct tty_struct tty_table[]; // tty 结构数组。
```

20. 文件状态结构 (include/sys/stat.h, 第 6 行)

```
struct stat {
    dev_t  st_dev;      // 含有文件的设备号。
    ino_t  st_ino;      // 文件 i 节点号。
    umode_t st_mode;    // 文件属性（见下面）。
    nlink_t st_nlink;   // 指定文件的连接数。
    uid_t  st_uid;      // 文件的用户(标识)号。
    gid_t  st_gid;      // 文件的组号。
    dev_t  st_rdev;     // 设备号(如果文件是特殊的字符文件或块文件)。
    off_t  st_size;     // 文件大小（字节数）（如果文件是常规文件）。
    time_t st_atime;    // 上次（最后）访问时间。
    time_t st_mtime;    // 最后修改时间。
    time_t st_ctime;    // 最后节点修改时间。
};
```

21. 文件访问与修改时间结构 (include/sys/times.h, 第 6 行)

```
struct tms {
    time_t tms_utime; // 用户使用的 CPU 时间。
    time_t tms_stime; // 系统（内核）CPU 时间。
    time_t tms_cutime; // 已终止的子进程使用的用户 CPU 时间。
    time_t tms_cstime; // 已终止的子进程使用的系统 CPU 时间。
};
```

};

22. ustat 结构 (include/sys/types.h, 第 39 行)

文件系统参数结构，用于函数 ustat()。

```
struct ustat {
    daddr\_t f_tfree;           // 系统总空闲块数。
    ino\_t f_tinode;           // 总空闲 i 节点数。
    char f_fname[6];          // 文件系统名称。
    char f_fpack[6];          // 文件系统压缩名称。
};
```

最后两个字段未使用，总是返回 NULL 指针。

23. 系统名称头文件 (include/sys/utsname.h, 第 6 行)

```
struct utsname {
    char sysname[9];           // 本版本操作系统的名称。
    char nodename[9];          // 与实现相关的网络中节点名称。
    char release[9];           // 本实现的当前发行级别。
    char version[9];           // 本次发行的版本级别。
    char machine[9];           // 系统运行的硬件类型名称。
};
```

24. 块设备请求项结构 (kernel/blk_drv/blk.h, 第 23 行)

下面是请求队列中项的结构。其中如果 dev=-1，则表示没有使用该项。

```
struct request {
    int dev;                   // 使用的设备号，未用时为-1。
    int cmd;                   // 命令(READ 或 WRITE)。
    int errors;                // 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。
};
```

附录2 ASCII 码表

十进制	十六进制	字符	十进制	十六进制	字符	十进制	十六进制	字符
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	\
7	07	BEL	50	32	2	93	5D]
8	08	BS	51	33	3	94	5E	^
9	09	TAB	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	`
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	(space)	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	\$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	
39	27	'	82	52	R	125	7D	}
40	28	(83	53	S	126	7E	~
41	29)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

附录3 常用 C0、C1 控制字符表

常用 C0 控制字符表

助记符	代码值	采取的行动
NUL	0x00	Null -- 在接收到时忽略（不保存在输入缓冲中）。
ENQ	0x05	Enquiry -- 传送应答消息。
BEL	0x07	Bell -- 发声响。
BS	0x08	Backspace -- 将光标左移一个字符。若光标已经处在左边沿，则无动作。
HT	0x09	Horizontal Tabulation -- 将光标移到下一个制表位。若右侧已经没有制表位，则移到右边缘处。
LF	0x0a	Linefeed -- 此代码导致一个回车或换行操作（见换行模式）。
VT	0x0b	Virtual Tabulation -- 作用如 LF。
FF	0x0c	Form Feed -- 作用如 LF。
CR	0x0d	Carriage Return -- 将光标移到当前行的左边缘处。
SO	0x0e	Shift Out -- 使用由 SCS 控制序列选择的 G1 字符集。G1 可指定 5 种字符集之一。
SI	0x0f	Shift In -- 使用由 SCS 控制序列选择的 G0 字符集。G0 可指定 5 种字符集之一。
DC1	0x11	Device Control 1 -- 即 XON。使终端重新继续传输。
DC3	0x13	Device Control 3 -- 即 XOFF。使中断除发送 XOFF 和 XON 外，停止发送其他所有代码。
CAN	0x18	Cancel -- 若在控制序列期间发送，则序列不会执行而立刻终止。同时显示出错字符。
SUB	0x1a	Substitute -- 作用同 CAN。
ESC	0x1b	Escape -- 产生一个转义控制序列。
DEL	0x7f	Delete -- 在输入时忽略（不保存在输入缓冲中）。

常用 C1 控制字符表。

助记符	代码值	7B 表示	采取的行动
IND	0x84	ESC D	Index -- 光标在同列下移一行。若光标已在底行，则执行滚屏操作。
NEL	0x85	ESC H	Next Line -- 光标移动到下一行头一列。若光标已在底行，则执行滚屏操作。
HTS	0x88	ESC E	Horizontal Tab Set -- 在光标处设置一个水平制表位。
RI	0x8d	ESC M	Reverse index -- 光标在同列上移一行。若光标已在顶行，则执行滚屏操作。
SS2	0x8e	ESC N	Single Shift G2 -- 为显示下一个字符临时调用 GL 中 G2 字符集。G2 由选择字符集（SCS）控制序列指定（参见下面转义序列和控制序列表）。
SS3	0x8f	ESC O	Single Shift G3 -- 为显示下一个字符临时调用 GL 中 G3 字符集。G3 由选择字符集（SCS）控制序列指定（参见下面转义序列和控制序列表）。
DCS	0x90	ESC P	Device Control String -- 作为设备控制字符串的起始限定符。
CSI	0x9b	ESC [Control Sequence Introducer -- 作为控制序列引导符。
ST	0x9c	ESC \	String Terminator -- 作为 DCS 串的结尾限定符。

附录4 常用转义序列和控制序列

序列和名称	说明
ESC (Ps 和 ESC) Ps 选择字符集	Select Character Set (SCS) -- G0 和 G1 字符集可以分别指定 5 种字符集之一。'ESC (Ps' 指定 G0 所用的字符集，'ESC) Ps' 指定 G1 所用的字符集。参数 Ps: A - UK 字符集; B - US 字符集; 0 - 图形字符集; 1 - 另选 ROM 字符集; 2 - 另选 ROM 特殊字符集。 一个终端可以显示最多 254 个不同的字符，然而终端只在其 ROM 中保存了 127 个显示字符。你必须为其他 127 个显示字符安装另外的字符集 ROM。在某一个时刻，终端能够选择 94 个字符（一个字符集）。因此，终端可以使用五个字符集之一，其中有些字符出现在多个字符集中。在任一时刻，终端可以使用两个活动字符集。计算机可以使用 SCS 序列把任意两个字符集指定为 G0 和 G1。此后使用单个控制字符就可以在这两个字符集之间进行切换。换进（Shift In - SI, 14）控制字符调入 G0 字符集，而换出（Shift Out - SO, 15）控制字符则可以调入 G1 字符集。指定的字符集呈现为当前使用字符集，直到终端收到另外一个 SCS 序列。
ESC [Pn A 光标上移 (终端↔主机)	Cursor Up (CUU) -- CUU 控制序列把光标上移但列位置不变。移动字符位置数由参数确定。如果参数是 Pn，则光标上移 Pn 行。光标最多上移到顶行。注意，Pn 是一个 ASCII 码数字变量。如果你没有选择参数或参数值为 0，那么终端将假定参数值为 1。
ESC [Pn B 或 ESC [Pn e 光标下移 (终端↔主机)	Cursor Down (CUD) -- CUD 控制序列把光标下移但列位置不变。移动字符位置数由参数确定。如果参数是 1 或 0，则光标下移 1 行。如果参数是 Pn，则光标下移 Pn 行。光标最多下移到底行。
ESC [Pn C 或 ESC [Pn a 光标右移 (终端↔主机)	Cursor Forward (CUF) -- CUF 控制序列把当前光标向右移动。移动位置数由参数确定。如果参数是 1 或 0，则移动 1 个字符位置。如果参数值是 Pn，则光标移动 Pn 个字符位置。光标最多移动到右边界。
ESC [Pn D 光标左移 (终端↔主机)	Cursor Backward (CUB) -- CUB 控制序列把当前光标向左移动。移动位置数由参数确定。如果参数是 1 或 0，则移动 1 个字符位置。如果参数值是 Pn，则光标移动 Pn 个字符位置。光标最多移动到左边界。
ESC [Pn E 光标向下移动	Cursor Next Line (CNL) -- 该控制序列把光标移动到下面第 Pn 行第 1 个字符上。
ESC [Pn F 光标向上移动	Cursor Last Line (CLL) -- 该控制序列把光标向上移动到第 Pn 行第 1 个字符上。
ESC [Pn G 或 ESC [Pn ` 光标水平移动	Cursor Horizon Absolute (CHA) -- 该控制序列把光标移动到当前行第 Pn 个字符处。
ESC [Pn ; Pn H 或 ESC [Pn ; Pn f 光标定位	Cursor Position (CUP), Horizontal And Vertical Position(HVP) -- CUP 控制序列把当前光标移动到参数指定的位置处。两个参数分别指定行、列值。若值为 0 则同 1，表示移动 1 个位置。在不含参数的默认条件下等同于把光标移动到 home 位置（即 ESC [H）。
ESC [Pn d 设置行位置	Vertical Line Position Absolute -- 将光标移动到当前行的 Pn 行处。如果试图移动到最后一行下面，那么光标将停留在最后一行上。
ESC [s 保存光标位置	Save Current Cursor Position -- 该控制序列与 DECSC 作用相同，除了光标所处显示页页号并不会保存。
ESC [u 恢复光标位置	Restore Saved Cursor Position -- 该控制序列与 DECRC 作用相同，除了光标仍然处于同一显示页面而非移动到光标被保存的显示页。
ESC D	Index (IND) -- 该控制序列使得光标下移 1 行，但列号不变。如果光标正处于底行，则会导致

索引	屏幕向上滚动 1 行。
ESC M 反向索引	Reverse Index (RI) -- 该控制序列是的光标上移 1 行，但列号不变。如果光标正处于顶行，则会导致屏幕向下滚动 1 行。
ESC E 下移 1 行	Next Line (NEL) -- 该控制序列将使得光标移动到下一行的左边开始处。如果光标正处于底行，则会导致屏幕向上滚动 1 行。
ESC 7 保存光标	Save Cursor (DECSO) -- 这个控制序列将导致光标位置、图形重现以及字符集被保存。
ESC 8 恢复光标	Restore Cursor (DECRC) -- 这个控制序列将导致先前保存的光标位置、图形重现以及字符集被恢复重置。
ESC [Ps; Ps; ... ; Ps m 设置字符属性	Select Graphic Rendition (SGR) -- 字符重显与属性是不改变字符代码前提下影响一个字符显示方式的特性。该控制序列根据参数设置字符显示属性。以后所有发送到终端的字符都将使用这里指定的属性，直到再次执行本控制序列重新设置字符显示的属性。参数 Ps: 0 - 无属性（默认属性）；1 - 粗体并增亮；4 - 下划线；5 - 闪烁；7 - 反显；22 - 非粗体；24 - 无下划线；25 - 无闪烁；27 - 正显；30--38 设置前景色彩；39 - 默认前景色（White）；40--48 - 设置背景色彩；49 - 默认背景色（Black）。30--37 和 40--47 分别对应颜色：Black、Red、Green、Yellow、Blue、Magenta、Cyan、White。
ESC [Pn L 插入行	Insert Line (IL) -- 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。
ESC [Pn M 删除行	Delete Line (DL) -- 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。
ESC [Pn @ 插入字符	Insert Character (ICH) -- 该控制序列使用普通字符属性在当前光标处插入 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。
ESC [Pn P 删除光标处字符	Delete Character (DCH) -- 该控制序列从光标处删除 Pn 个字符。当一个字符被删除时，光标右面的所有字符都左移。这会在右边界处产生一个空字符。其属性与最后一个左移字符相同。
ESC [Ps J 擦除字符	Erase In Display (ED) -- 根据参数，该控制序列擦除部分或所有显示的字符。擦除操作从屏幕上移走字符但不影响其他字符。擦除的字符被丢弃。在擦除字符或行时光标位置不变。在擦除字符的同时，字符的属性也被丢弃。该控制序列擦除的任何整行将会把该行回置到单个字符宽度模式。参数 Ps: 0 - 擦除光标到屏幕底端所有字符；1 - 擦除屏幕顶端到光标处所有字符；2 - 擦除整屏。
ESC [Ps K 行内擦除	Erase In Line (EL) -- 根据参数擦除光标所在行的部分或所有字符。擦除操作从屏幕上移走字符但不影响其他字符。擦除的字符被丢弃。在擦除字符或行时光标位置不变。在擦除字符的同时，字符的属性也被丢弃。参数 Ps: 0 - 擦除光标到行末所有字符；1 - 擦除左边界到光标处所有字符；2 - 擦除一整行。
ESC [Pn ; Pn r 设置上下边界	Set Top and Bottom Margins (DECSTBM) -- 该控制序列设置卷屏上下区域。滚屏边界是屏幕上的一个区域，通过从屏幕上卷走原字符我们其中可以接收新的字符。该区域通过屏幕顶端和低端边界来定义。第 1 个参数是滚屏区域的开始第 1 行，第 2 个参数是滚屏区域的最后 1 行。默认情况下是整个屏幕。最小的滚屏区域是 2 行，即顶边界行必须小于底边界行。光标将被放置在 home 位置。
ESC [Pn c 或 ESC Z 设备属性 (终端 ↔ 主机)	为响应主机请求终端可以发送报告信息。这些信息提供了标识（终端类型）、光标位置和终端操作状态。共有两类报告：设备属性和设备状态报告。Device Attributes (DA) -- 主机通过发送不带参数或参数是 0 的 DA 控制序列要求终端发送一个设备属性（DA）控制序列（ESC Z 的作用与此相同），终端则发送以下序列之一来响应主机的序列：

附录 4 常用转义序列和控制序列

	<div>终端可选属性</div> <div>发送序列</div> <div>终端可选属性</div> <div>发送序列</div>
	<div>无，VT101</div> <div>ESC [?1;0c</div> <div>处理器选项（STP）</div> <div>ESC [?1;1c</div>
	<div>高级视频（AVO）VT100</div> <div>ESC [?1;2c</div> <div>AVO 和 STP</div> <div>ESC [?1;3c</div>
	<div>图形选项（GPO）</div> <div>ESC [?1;4c</div> <div>GPO 和 STP</div> <div>ESC [?1;5c</div>
	<div>GPO 和 AVO，VT102</div> <div>ESC [?1;6c</div> <div>GPO、STP 和 AVO</div> <div>ESC [?1;7c</div>
<div>ESC c</div> <div>复位到初始状态</div>	<div>Reset To Initial State (RIS) -- 让终端复位到其初始状态，即刚打开电源的状态。复位阶段接收的字符将全部丢失。可以采用两种方式避免：1.（自动 XON/XOFF）在发送之后主机假设终端发送了 XOFF。主机停止发送字符直到接收到 XON。2. 延迟起码 10 秒，等待终端复位操作完成。</div>

附录5 第 1 套键盘扫描码集

键 (KEY)	接通码 (MAKE)	断开码 (BREAK)	键 (KEY)	接通码 (MAKE)	断开码 (BREAK)	键 (KEY)	接通码 (MAKE)	断开码 (BREAK)
A	1E	9E	9	0A	8A	[1A	9A
B	30	B0	`	29	89	INSERT	E0, 52	E0, D2
C	2E	AE	-	0C	8C	HOME	E0, 47	E0, 97
D	20	A0	=	0D	8D	PG UP	E0, 49	E0, C9
E	12	92	\	2B	AB	DELETE	E0, 53	E0, D3
F	21	A1	BKSP	0E	8E	END	E0, 4F	E0, CF
G	22	A2	SPACE	39	B9	PG DN	E0, 51	E0, D1
H	23	A3	TAB	0F	8F	向上箭头	E0, 48	E0, C8
I	17	97	CAPS	3A	BA	向左箭头	E0, 4B	E0, CB
J	24	A4	左 SHFT	2A	AA	向下箭头	E0, 50	E0, D0
K	25	A5	左 CTRL	1D	9D	向右箭头	E0, 4D	E0, CD
L	26	A6	左 GUI	E0, 5B	E0, DB	NUM LOCK	45	C5
M	32	B2	左 ALT	38	B8	KP /	E0, 35	E0, B5
N	31	B1	右 SHFT	36	B6	KP *	37	B7
O	18	98	右 CTRL	E0, 1D	E0, 9D	KP -	4A	CA
P	19	99	右 GUI	E0, 5C	E0, DC	KP +	4E	CE
Q	10	90	右 ALT	E0, 38	E0, B8	KP ENTER	E0, 1C	E0, 9C
R	13	93	APPS	E0, 5D	E0, DD	KP .	53	D3
S	1F	9F	ENTER	1C	9C	KP 0	52	D2
T	14	94	ESC	01	81	KP 1	4F	CF
U	16	96	F1	3B	BB	KP 2	50	D0
V	2F	AF	F2	3C	BC	KP 3	51	D1
W	11	91	F3	3D	BD	KP 4	4B	CB
X	2D	AD	F4	3E	BE	KP 5	4C	CC
Y	15	95	F5	3F	BF	KP 6	4D	CD
Z	2C	AC	F6	40	C0	KP 7	47	C7
0	0B	8B	F7	41	C1	KP 8	48	C8
1	02	82	F8	42	C2	KP 9	49	C9
2	03	83	F9	43	C3]	1B	9B
3	04	84	F10	44	C4	;	27	A7
4	05	85	F11	57	D7	,	28	A8
5	06	86	F12	58	D8	,	33	B3
6	07	87	PRNT SCRN	E0, 2A, E0, 37	E0, B7, E0, AA	.	34	B4
7	08	88	SCROLL	46	C6	/	35	B5
8	09	89	PAUSE	E1, 1D, 45 E1, 9D, C5	无			

注 1：表中所有数值均为十六进制。

注 2：表中 KP -- KeyPad，表示数字小键盘上的键。

注 3：表中着色部分均为扩展按键。

索引

由于内核代码相对比较庞大，很多变量/函数在源代码的很多程序中被使用/调用，因此对其进行索引比较困难。本索引主要根据变量或函数名称给出定义它的程序文件名、行号和所在页码。

__strtok
include/string.h, 275, 定义为变量
__GNU_EXEC_MACROS__
include/a.out.h, 4, 定义为预处理宏
__LIBRARY__
init/main.c, 7, 定义为预处理宏
lib/close.c, 7, 定义为预处理宏
lib/dup.c, 7, 定义为预处理宏
lib/_exit.c, 7, 定义为预处理宏
lib/open.c, 7, 定义为预处理宏
lib/execve.c, 7, 定义为预处理宏
lib/setuid.c, 7, 定义为预处理宏
lib/string.c, 13, 定义为预处理宏
lib/wait.c, 7, 定义为预处理宏
lib/write.c, 7, 定义为预处理宏
__NR_access
include/unistd.h, 93, 定义为预处理宏
__NR_acct
include/unistd.h, 111, 定义为预处理宏
__NR_alarm
include/unistd.h, 87, 定义为预处理宏
__NR_break
include/unistd.h, 77, 定义为预处理宏
__NR_brk
include/unistd.h, 105, 定义为预处理宏
__NR_chdir
include/unistd.h, 72, 定义为预处理宏
__NR_chmod
include/unistd.h, 75, 定义为预处理宏
__NR_chown
include/unistd.h, 76, 定义为预处理宏
__NR_chroot
include/unistd.h, 121, 定义为预处理宏
__NR_close
include/unistd.h, 66, 定义为预处理宏
__NR_creat
include/unistd.h, 68, 定义为预处理宏
__NR_dup
include/unistd.h, 101, 定义为预处理宏
__NR_dup2
include/unistd.h, 123, 定义为预处理宏
__NR_execve
include/unistd.h, 71, 定义为预处理宏
__NR_exit
include/unistd.h, 61, 定义为预处理宏
__NR_fcntl
include/unistd.h, 115, 定义为预处理宏

__NR_fork
include/unistd.h, 62, 定义为预处理宏
__NR_fstat
include/unistd.h, 88, 定义为预处理宏
__NR_ftime
include/unistd.h, 95, 定义为预处理宏
__NR_getegid
include/unistd.h, 110, 定义为预处理宏
__NR_geteuid
include/unistd.h, 109, 定义为预处理宏
__NR_getgid
include/unistd.h, 107, 定义为预处理宏
__NR_getpgrp
include/unistd.h, 125, 定义为预处理宏
__NR_getpid
include/unistd.h, 80, 定义为预处理宏
__NR_getppid
include/unistd.h, 124, 定义为预处理宏
__NR_getuid
include/unistd.h, 84, 定义为预处理宏
__NR_gtty
include/unistd.h, 92, 定义为预处理宏
__NR_ioctl
include/unistd.h, 114, 定义为预处理宏
__NR_kill
include/unistd.h, 97, 定义为预处理宏
__NR_link
include/unistd.h, 69, 定义为预处理宏
__NR_lock
include/unistd.h, 113, 定义为预处理宏
__NR_lseek
include/unistd.h, 79, 定义为预处理宏
__NR_mkdir
include/unistd.h, 99, 定义为预处理宏
__NR_mknod
include/unistd.h, 74, 定义为预处理宏
__NR_mount
include/unistd.h, 81, 定义为预处理宏
__NR_mpx
include/unistd.h, 116, 定义为预处理宏
__NR_nice
include/unistd.h, 94, 定义为预处理宏
__NR_open
include/unistd.h, 65, 定义为预处理宏
__NR_pause
include/unistd.h, 89, 定义为预处理宏
__NR_phys
include/unistd.h, 112, 定义为预处理宏

__NR_pipe
 include/unistd.h, 102, 定义为预处理宏
 __NR_prof
 include/unistd.h, 104, 定义为预处理宏
 __NR_ptrace
 include/unistd.h, 86, 定义为预处理宏
 __NR_read
 include/unistd.h, 63, 定义为预处理宏
 __NR_rename
 include/unistd.h, 98, 定义为预处理宏
 __NR_rmdir
 include/unistd.h, 100, 定义为预处理宏
 __NR_setgid
 include/unistd.h, 106, 定义为预处理宏
 __NR_setpgid
 include/unistd.h, 117, 定义为预处理宏
 __NR_setregid
 include/unistd.h, 131, 定义为预处理宏
 __NR_setreuid
 include/unistd.h, 130, 定义为预处理宏
 __NR_setsid
 include/unistd.h, 126, 定义为预处理宏
 __NR_setuid
 include/unistd.h, 83, 定义为预处理宏
 __NR_setup
 include/unistd.h, 60, 定义为预处理宏
 __NR_sgetmask
 include/unistd.h, 128, 定义为预处理宏
 __NR_sigaction
 include/unistd.h, 127, 定义为预处理宏
 __NR_signal
 include/unistd.h, 108, 定义为预处理宏
 __NR_ssetmask
 include/unistd.h, 129, 定义为预处理宏
 __NR_stat
 include/unistd.h, 78, 定义为预处理宏
 __NR_stime
 include/unistd.h, 85, 定义为预处理宏
 __NR_stty
 include/unistd.h, 91, 定义为预处理宏
 __NR_sync
 include/unistd.h, 96, 定义为预处理宏
 __NR_time
 include/unistd.h, 73, 定义为预处理宏
 __NR_times
 include/unistd.h, 103, 定义为预处理宏
 __NR_ulimit
 include/unistd.h, 118, 定义为预处理宏
 __NR_umask
 include/unistd.h, 120, 定义为预处理宏
 __NR_umount
 include/unistd.h, 82, 定义为预处理宏
 __NR_uname
 include/unistd.h, 119, 定义为预处理宏
 __NR_unlink
 include/unistd.h, 70, 定义为预处理宏
 __NR_ustat

include/unistd.h, 122, 定义为预处理宏
 __NR_utime
 include/unistd.h, 90, 定义为预处理宏
 __NR_waitpid
 include/unistd.h, 67, 定义为预处理宏
 __NR_write
 include/unistd.h, 64, 定义为预处理宏
 __va_rounded_size
 include/stdarg.h, 9, 定义为预处理宏
 _A_OUT_H
 include/a.out.h, 2, 定义为预处理宏
 _BLK_H
 kernel/blk_drv/blk.h, 2, 定义为预处理宏
 _BLOCKABLE
 kernel/sched.c, 24, 定义为预处理宏
 _bmap
 fs/inode.c, 72, 定义为函数
 _bucket_dir
 lib/malloc.c, 60, 定义为struct类型
 _C
 include/ctype.h, 7, 定义为预处理宏
 _CONFIG_H
 include/config.h, 2, 定义为预处理宏
 _CONST_H
 include/const.h, 2, 定义为预处理宏
 _ctmp
 include/ctype.h, 14, 定义为变量
 lib/ctype.c, 9, 定义为变量
 _ctype
 include/ctype.h, 13, 定义为变量
 lib/ctype.c, 10, 定义为变量
 _CTYPE_H
 include/ctype.h, 2, 定义为预处理宏
 _D
 include/ctype.h, 6, 定义为预处理宏
 _ERRNO_H
 include/errno.h, 2, 定义为预处理宏
 _exit
 include/unistd.h, 208, 定义为函数原型
 lib/_exit.c, 10, 定义为函数
 _FCNTL_H
 include/fcntl.h, 2, 定义为预处理宏
 _FDREG_H
 include/fdreg.h, 7, 定义为预处理宏
 _fs
 kernel/traps.c, 34, 定义为预处理宏
 _FS_H
 include/fs.h, 7, 定义为预处理宏
 _get_base
 include/sched.h, 214, 定义为预处理宏
 _hashfn
 fs/buffer.c, 128, 定义为预处理宏
 _HDREG_H
 include/hdreg.h, 7, 定义为预处理宏
 _HEAD_H
 include/head.h, 2, 定义为预处理宏

- `_HIGH`
- `include/sys/wait.h`, 7, 定义为预处理宏
- `_I_FLAG`
- `kernel/chr_drv/tty_io.c`, 29, 定义为预处理宏
- `_L`
- `include/ctype.h`, 5, 定义为预处理宏
- `_L_FLAG`
- `kernel/chr_drv/tty_io.c`, 28, 定义为预处理宏
- `_LDT`
- `include/sched.h`, 156, 定义为预处理宏
- `_LOW`
- `include/sys/wait.h`, 6, 定义为预处理宏
- `_MM_H`
- `include/mm.h`, 2, 定义为预处理宏
- `_N_BADMAG`
- `include/a.out.h`, 36, 定义为预处理宏
- `_N_HDROFF`
- `include/a.out.h`, 40, 定义为预处理宏
- `_N_SEGMENT_ROUND`
- `include/a.out.h`, 95, 定义为预处理宏
- `_N_TXTENDADDR`
- `include/a.out.h`, 97, 定义为预处理宏
- `_NSIG`
- `include/signal.h`, 9, 定义为预处理宏
- `_O_FLAG`
- `kernel/chr_drv/tty_io.c`, 30, 定义为预处理宏
- `_P`
- `include/ctype.h`, 8, 定义为预处理宏
- `_PC_CHOWN_RESTRICTED`
- `include/unistd.h`, 51, 定义为预处理宏
- `_PC_LINK_MAX`
- `include/unistd.h`, 43, 定义为预处理宏
- `_PC_MAX_CANON`
- `include/unistd.h`, 44, 定义为预处理宏
- `_PC_MAX_INPUT`
- `include/unistd.h`, 45, 定义为预处理宏
- `_PC_NAME_MAX`
- `include/unistd.h`, 46, 定义为预处理宏
- `_PC_NO_TRUNC`
- `include/unistd.h`, 49, 定义为预处理宏
- `_PC_PATH_MAX`
- `include/unistd.h`, 47, 定义为预处理宏
- `_PC_PIPE_BUF`
- `include/unistd.h`, 48, 定义为预处理宏
- `_PC_VDISABLE`
- `include/unistd.h`, 50, 定义为预处理宏
- `_POSIX_CHOWN_RESTRICTED`
- `include/unistd.h`, 7, 定义为预处理宏
- `_POSIX_NO_TRUNC`
- `include/unistd.h`, 8, 定义为预处理宏
- `_POSIX_VDISABLE`
- `include/unistd.h`, 9, 定义为预处理宏
- `_POSIX_VERSION`
- `include/unistd.h`, 5, 定义为预处理宏
- `_PTRDIFF_T`
- `include/sys/types.h`, 15, 定义为预处理宏
- `include/stddef.h`, 5, 定义为预处理宏
- `_S`
- `include/ctype.h`, 9, 定义为预处理宏
- `kernel/sched.c`, 23, 定义为预处理宏
- `_SC_ARG_MAX`
- `include/unistd.h`, 33, 定义为预处理宏
- `_SC_CHILD_MAX`
- `include/unistd.h`, 34, 定义为预处理宏
- `_SC_CLOCKS_PER_SEC`
- `include/unistd.h`, 35, 定义为预处理宏
- `_SC_JOB_CONTROL`
- `include/unistd.h`, 38, 定义为预处理宏
- `_SC_NGROUPS_MAX`
- `include/unistd.h`, 36, 定义为预处理宏
- `_SC_OPEN_MAX`
- `include/unistd.h`, 37, 定义为预处理宏
- `_SC_SAVED_IDS`
- `include/unistd.h`, 39, 定义为预处理宏
- `_SC_VERSION`
- `include/unistd.h`, 40, 定义为预处理宏
- `_SCHED_H`
- `include/sched.h`, 2, 定义为预处理宏
- `_set_base`
- `include/sched.h`, 188, 定义为预处理宏
- `_set_gate`
- `include/asm/system.h`, 22, 定义为预处理宏
- `_set_limit`
- `include/sched.h`, 199, 定义为预处理宏
- `_set_seg_desc`
- `include/asm/system.h`, 42, 定义为预处理宏
- `_set_tssldt_desc`
- `include/asm/system.h`, 52, 定义为预处理宏
- `_SIGNAL_H`
- `include/signal.h`, 2, 定义为预处理宏
- `_SIZE_T`
- `include/sys/types.h`, 5, 定义为预处理宏
- `include/time.h`, 10, 定义为预处理宏
- `include/stddef.h`, 10, 定义为预处理宏
- `include/string.h`, 9, 定义为预处理宏
- `_SP`
- `include/ctype.h`, 11, 定义为预处理宏
- `_STDARG_H`
- `include/stdarg.h`, 2, 定义为预处理宏
- `_STDDEF_H`
- `include/stddef.h`, 2, 定义为预处理宏
- `_STRING_H`
- `include/string.h`, 2, 定义为预处理宏
- `_SYS_STAT_H`
- `include/sys/stat.h`, 2, 定义为预处理宏
- `_SYS_TYPES_H`
- `include/sys/types.h`, 2, 定义为预处理宏
- `_SYS_UTSNAME_H`
- `include/sys/utsname.h`, 2, 定义为预处理宏
- `_SYS_WAIT_H`
- `include/sys/wait.h`, 2, 定义为预处理宏
- `_syscall0`

- include/unistd.h, 133, 定义为预处理宏
 _syscall1
 include/unistd.h, 146, 定义为预处理宏
 _syscall2
 include/unistd.h, 159, 定义为预处理宏
 _syscall3
 include/unistd.h, 172, 定义为预处理宏
 _TERMIOS_H
 include/termios.h, 2, 定义为预处理宏
 _TIME_H
 include/time.h, 2, 定义为预处理宏
 _TIME_T
 include/sys/types.h, 10, 定义为预处理宏
 include/time.h, 5, 定义为预处理宏
 _TIMES_H
 include/sys/times.h, 2, 定义为预处理宏
 _TSS
 include/sched.h, 155, 定义为预处理宏
 _TTY_H
 include/tty.h, 10, 定义为预处理宏
 _U
 include/ctype.h, 4, 定义为预处理宏
 _UNISTD_H
 include/unistd.h, 2, 定义为预处理宏
 _UTIME_H
 include/utime.h, 2, 定义为预处理宏
 _X
 include/ctype.h, 10, 定义为预处理宏
 ABRT_ERR
 include/hdreg.h, 47, 定义为预处理宏
 ACC_MODE
 fs/namei.c, 21, 定义为预处理宏
 access
 include/unistd.h, 189, 定义为函数原型
 acct
 include/unistd.h, 190, 定义为函数原型
 add_entry
 fs/namei.c, 165, 定义为函数
 add_request
 kernel/blk_drv/ll_rw_blk.c, 64, 定义为函数
 add_timer
 include/sched.h, 144, 定义为函数原型
 kernel/sched.c, 272, 定义为函数
 alarm
 include/unistd.h, 191, 定义为函数原型
 ALRMMASK
 kernel/chr_drv/tty_io.c, 17, 定义为预处理宏
 argv
 init/main.c, 165, 定义为变量
 argv_rc
 init/main.c, 162, 定义为变量
 asctime
 include/time.h, 35, 定义为函数原型
 attr
 kernel/chr_drv/console.c, 77, 定义为变量
 B0
 include/termios.h, 133, 定义为预处理宏
 B110
 include/termios.h, 136, 定义为预处理宏
 B1200
 include/termios.h, 142, 定义为预处理宏
 B134
 include/termios.h, 137, 定义为预处理宏
 B150
 include/termios.h, 138, 定义为预处理宏
 B1800
 include/termios.h, 143, 定义为预处理宏
 B19200
 include/termios.h, 147, 定义为预处理宏
 B200
 include/termios.h, 139, 定义为预处理宏
 B2400
 include/termios.h, 144, 定义为预处理宏
 B300
 include/termios.h, 140, 定义为预处理宏
 B38400
 include/termios.h, 148, 定义为预处理宏
 B4800
 include/termios.h, 145, 定义为预处理宏
 B50
 include/termios.h, 134, 定义为预处理宏
 B600
 include/termios.h, 141, 定义为预处理宏
 B75
 include/termios.h, 135, 定义为预处理宏
 B9600
 include/termios.h, 146, 定义为预处理宏
 bad_flp_intr
 kernel/blk_drv/floppy.c, 233, 定义为函数
 bad_rw_intr
 kernel/blk_drv/hd.c, 242, 定义为函数
 BADNESS
 fs/buffer.c, 205, 定义为预处理宏
 BBD_ERR
 include/hdreg.h, 50, 定义为预处理宏
 BCD_TO_BIN
 init/main.c, 74, 定义为预处理宏
 beepcount
 kernel/chr_drv/console.c, 697, 定义为变量
 blk_dev
 kernel/blk_drv/ll_rw_blk.c, 32, 定义为struct类型
 kernel/blk_drv/blk.h, 50, 定义为struct类型
 blk_dev_init
 init/main.c, 46, 定义为函数原型
 kernel/blk_drv/ll_rw_blk.c, 157, 定义为函数
 blk_dev_struct
 kernel/blk_drv/blk.h, 45, 定义为struct类型
 block_read
 fs/read_write.c, 18, 定义为函数原型
 fs/block_dev.c, 47, 定义为函数
 BLOCK_SIZE
 include/fs.h, 49, 定义为预处理宏

- BLOCK_SIZE_BITS
include/fs.h, 50, 定义为预处理宏
block_write
fs/read_write.c, 19, 定义为函数原型
fs/block_dev.c, 14, 定义为函数
bmap
fs/inode.c, 140, 定义为函数
include/fs.h, 176, 定义为函数原型
bottom
kernel/chr_drv/console.c, 73, 定义为变量
bounds
kernel/traps.c, 48, 定义为函数原型
bread
fs/buffer.c, 267, 定义为函数
include/fs.h, 189, 定义为函数原型
bread_page
fs/buffer.c, 296, 定义为函数
include/fs.h, 190, 定义为函数原型
breada
fs/buffer.c, 322, 定义为函数
include/fs.h, 191, 定义为函数原型
brelease
fs/buffer.c, 253, 定义为函数
include/fs.h, 188, 定义为函数原型
brk
include/unistd.h, 192, 定义为函数原型
BRKINT
include/termios.h, 84, 定义为预处理宏
BS0
include/termios.h, 122, 定义为预处理宏
BS1
include/termios.h, 123, 定义为预处理宏
BSDLY
include/termios.h, 121, 定义为预处理宏
bucket_desc
lib/malloc.c, 52, 定义为struct类型
bucket_dir
lib/malloc.c, 77, 定义为变量
buffer_block
include/fs.h, 66, 定义为类型
BUFFER_END
include/const.h, 4, 定义为预处理宏
buffer_head
include/fs.h, 68, 定义为struct类型
buffer_init
fs/buffer.c, 348, 定义为函数
include/fs.h, 31, 定义为函数原型
buffer_memory_end
init/main.c, 99, 定义为变量
buffer_wait
fs/buffer.c, 33, 定义为变量
BUSY_STAT
include/hdreg.h, 31, 定义为预处理宏
calc_mem
mm/memory.c, 413, 定义为函数
CBAUD
include/termios.h, 132, 定义为预处理宏
cfgetispeed
include/termios.h, 216, 定义为函数原型
cfgetospeed
include/termios.h, 217, 定义为函数原型
cfsetispeed
include/termios.h, 218, 定义为函数原型
cfsetospeed
include/termios.h, 219, 定义为函数原型
change_ldt
fs/exec.c, 154, 定义为函数
change_speed
kernel/chr_drv/tty_ioctl.c, 24, 定义为函数
CHARS
include/tty.h, 30, 定义为预处理宏
chdir
include/unistd.h, 194, 定义为函数原型
check_disk_change
fs/buffer.c, 113, 定义为函数
include/fs.h, 168, 定义为函数原型
chmod
include/sys/stat.h, 51, 定义为函数原型
include/unistd.h, 195, 定义为函数原型
chown
include/unistd.h, 196, 定义为函数原型
chr_dev_init
init/main.c, 47, 定义为函数原型
kernel/chr_drv/tty_io.c, 347, 定义为函数
chroot
include/unistd.h, 197, 定义为函数原型
CIBAUD
include/termios.h, 162, 定义为预处理宏
clear_bit
fs/bitmap.c, 25, 定义为预处理宏
clear_block
fs/bitmap.c, 13, 定义为预处理宏
cli
include/asm/system.h, 17, 定义为预处理宏
CLOCAL
include/termios.h, 161, 定义为预处理宏
clock
include/time.h, 30, 定义为函数原型
clock_t
include/time.h, 16, 定义为类型
CLOCKS_PER_SEC
include/time.h, 14, 定义为预处理宏
close
include/unistd.h, 198, 定义为函数原型
CMOS_READ
init/main.c, 69, 定义为预处理宏
kernel/blk_drv/hd.c, 28, 定义为预处理宏
CODE_SPACE
mm/memory.c, 49, 定义为预处理宏
command
kernel/blk_drv/floppy.c, 121, 定义为变量
con_init

- include/tty.h, 66, 定义为函数原型
- kernel/chr_drv/console.c, 617, 定义为函数
- con_write
- include/tty.h, 73, 定义为函数原型
- kernel/chr_drv/console.c, 445, 定义为函数
- controller_ready
- kernel/blk_drv/hd.c, 161, 定义为函数
- coprocessor_error
- kernel/traps.c, 58, 定义为函数原型
- coprocessor_segment_overrun
- kernel/traps.c, 52, 定义为函数原型
- copy_buffer
- kernel/blk_drv/floppy.c, 155, 定义为预处理宏
- copy_mem
- kernel/fork.c, 39, 定义为函数
- copy_page
- mm/memory.c, 54, 定义为预处理宏
- copy_page_tables
- include/sched.h, 29, 定义为函数原型
- mm/memory.c, 150, 定义为函数
- copy_process
- kernel/fork.c, 68, 定义为函数
- copy_strings
- fs/exec.c, 104, 定义为函数
- copy_to_cooked
- include/tty.h, 75, 定义为函数原型
- kernel/chr_drv/tty_io.c, 145, 定义为函数
- COPYBLK
- fs/buffer.c, 283, 定义为预处理宏
- cp_stat
- fs/stat.c, 15, 定义为函数
- CPARENB
- include/termios.h, 158, 定义为预处理宏
- CPARODD
- include/termios.h, 159, 定义为预处理宏
- cr
- kernel/chr_drv/console.c, 224, 定义为函数
- CR0
- include/termios.h, 111, 定义为预处理宏
- CR1
- include/termios.h, 112, 定义为预处理宏
- CR2
- include/termios.h, 113, 定义为预处理宏
- CR3
- include/termios.h, 114, 定义为预处理宏
- CRDLY
- include/termios.h, 110, 定义为预处理宏
- CREAD
- include/termios.h, 157, 定义为预处理宏
- creat
- include/unistd.h, 199, 定义为函数原型
- include/fcntl.h, 51, 定义为函数原型
- create_block
- fs/inode.c, 145, 定义为函数
- include/fs.h, 177, 定义为函数原型
- create_tables
- fs/exec.c, 46, 定义为函数
- CRTSCTS
- include/termios.h, 163, 定义为预处理宏
- crw_ptr
- fs/char_dev.c, 19, 定义为类型
- crw_table
- fs/char_dev.c, 85, 定义为变量
- CS5
- include/termios.h, 152, 定义为预处理宏
- CS6
- include/termios.h, 153, 定义为预处理宏
- CS7
- include/termios.h, 154, 定义为预处理宏
- CS8
- include/termios.h, 155, 定义为预处理宏
- csi_at
- kernel/chr_drv/console.c, 391, 定义为函数
- csi_J
- kernel/chr_drv/console.c, 239, 定义为函数
- csi_K
- kernel/chr_drv/console.c, 268, 定义为函数
- csi_L
- kernel/chr_drv/console.c, 401, 定义为函数
- csi_m
- kernel/chr_drv/console.c, 299, 定义为函数
- csi_M
- kernel/chr_drv/console.c, 421, 定义为函数
- csi_P
- kernel/chr_drv/console.c, 411, 定义为函数
- CSIZE
- include/termios.h, 151, 定义为预处理宏
- CSTOPB
- include/termios.h, 156, 定义为预处理宏
- ctime
- include/time.h, 36, 定义为函数原型
- cur_rate
- kernel/blk_drv/floppy.c, 113, 定义为变量
- cur_spec1
- kernel/blk_drv/floppy.c, 112, 定义为变量
- CURRENT
- kernel/blk_drv/blk.h, 93, 定义为预处理宏
- CURRENT_DEV
- kernel/blk_drv/blk.h, 94, 定义为预处理宏
- current_DOR
- kernel/sched.c, 204, 定义为变量
- kernel/blk_drv/floppy.c, 48, 定义为变量
- current_drive
- kernel/blk_drv/floppy.c, 115, 定义为变量
- CURRENT_TIME
- include/sched.h, 142, 定义为预处理宏
- current_track
- kernel/blk_drv/floppy.c, 120, 定义为变量
- d_inode
- include/fs.h, 83, 定义为struct类型
- d_super_block
- include/fs.h, 146, 定义为struct类型

daddr_t
 include/sys/types.h, 31, 定义为类型
 DAY
 kernel/mktime.c, 22, 定义为预处理宏
 debug
 kernel/traps.c, 44, 定义为函数原型
 DEC
 include/tty.h, 25, 定义为预处理宏
 DEFAULT_MAJOR_ROOT
 tools/build.c, 37, 定义为预处理宏
 DEFAULT_MINOR_ROOT
 tools/build.c, 38, 定义为预处理宏
 del
 kernel/chr_drv/console.c, 230, 定义为函数
 delete_char
 kernel/chr_drv/console.c, 363, 定义为函数
 delete_line
 kernel/chr_drv/console.c, 378, 定义为函数
 desc_struct
 include/head.h, 4, 定义为struct类型
 desc_table
 include/head.h, 6, 定义为类型
 dev_t
 include/sys/types.h, 26, 定义为类型
 DEVICE_INTR
 kernel/blk_drv/blk.h, 72, 定义为预处理宏
 kernel/blk_drv/blk.h, 81, 定义为预处理宏
 kernel/blk_drv/blk.h, 97, 定义为函数原型
 DEVICE_NAME
 kernel/blk_drv/blk.h, 63, 定义为预处理宏
 kernel/blk_drv/blk.h, 71, 定义为预处理宏
 kernel/blk_drv/blk.h, 80, 定义为预处理宏
 device_not_available
 kernel/traps.c, 50, 定义为函数原型
 DEVICE_NR
 kernel/blk_drv/blk.h, 65, 定义为预处理宏
 kernel/blk_drv/blk.h, 74, 定义为预处理宏
 kernel/blk_drv/blk.h, 83, 定义为预处理宏
 DEVICE_OFF
 kernel/blk_drv/blk.h, 67, 定义为预处理宏
 kernel/blk_drv/blk.h, 76, 定义为预处理宏
 kernel/blk_drv/blk.h, 85, 定义为预处理宏
 DEVICE_ON
 kernel/blk_drv/blk.h, 66, 定义为预处理宏
 kernel/blk_drv/blk.h, 75, 定义为预处理宏
 kernel/blk_drv/blk.h, 84, 定义为预处理宏
 DEVICE_REQUEST
 kernel/blk_drv/blk.h, 64, 定义为预处理宏
 kernel/blk_drv/blk.h, 73, 定义为预处理宏
 kernel/blk_drv/blk.h, 82, 定义为预处理宏
 kernel/blk_drv/blk.h, 99, 定义为函数原型
 die
 kernel/traps.c, 63, 定义为函数
 tools/build.c, 46, 定义为函数
 difftime
 include/time.h, 32, 定义为函数原型

DIR_ENTRIES_PER_BLOCK
 include/fs.h, 56, 定义为预处理宏
 dir_entry
 include/fs.h, 157, 定义为struct类型
 dir_namei
 fs/namei.c, 278, 定义为函数
 div_t
 include/sys/types.h, 36, 定义为类型
 divide_error
 kernel/traps.c, 43, 定义为函数原型
 DMA_READ
 include/fdreg.h, 68, 定义为预处理宏
 DMA_WRITE
 include/fdreg.h, 69, 定义为预处理宏
 do_bounds
 kernel/traps.c, 134, 定义为函数
 do_coprocessor_error
 kernel/traps.c, 169, 定义为函数
 do_coprocessor_segment_overrun
 kernel/traps.c, 149, 定义为函数
 do_debug
 kernel/traps.c, 124, 定义为函数
 do_device_not_available
 kernel/traps.c, 144, 定义为函数
 do_div
 kernel/vsprintf.c, 35, 定义为预处理宏
 do_divide_error
 kernel/traps.c, 97, 定义为函数
 do_double_fault
 kernel/traps.c, 87, 定义为函数
 do_execve
 fs/exec.c, 182, 定义为函数
 do_exit
 kernel/exit.c, 102, 定义为函数
 kernel/traps.c, 39, 定义为函数原型
 kernel/signal.c, 13, 定义为函数原型
 mm/memory.c, 31, 定义为函数原型
 do_fd_request
 kernel/blk_drv/floppy.c, 417, 定义为函数
 do_floppy_timer
 kernel/sched.c, 245, 定义为函数
 do_general_protection
 kernel/traps.c, 92, 定义为函数
 do_hd_request
 kernel/blk_drv/hd.c, 294, 定义为函数
 do_int3
 kernel/traps.c, 102, 定义为函数
 do_invalid_op
 kernel/traps.c, 139, 定义为函数
 do_invalid_TSS
 kernel/traps.c, 154, 定义为函数
 do_nmi
 kernel/traps.c, 119, 定义为函数
 do_no_page
 mm/memory.c, 365, 定义为函数
 do_overflow

- kernel/traps.c, 129, 定义为函数
do_rd_request
kernel/blk_drv/ramdisk.c, 23, 定义为函数
do_reserved
kernel/traps.c, 176, 定义为函数
do_segment_not_present
kernel/traps.c, 159, 定义为函数
do_signal
kernel/signal.c, 82, 定义为函数
do_stack_segment
kernel/traps.c, 164, 定义为函数
do_timer
kernel/sched.c, 305, 定义为函数
do_tty_interrupt
kernel/chr_drv/tty_io.c, 342, 定义为函数
do_wp_page
mm/memory.c, 247, 定义为函数
double_fault
kernel/traps.c, 51, 定义为函数原型
DRIVE
kernel/blk_drv/floppy.c, 54, 定义为预处理宏
drive_busy
kernel/blk_drv/hd.c, 202, 定义为函数
DRIVE_INFO
init/main.c, 59, 定义为预处理宏
drive_info
init/main.c, 102, 定义为struct类型
DRQ_STAT
include/hdreg.h, 27, 定义为预处理宏
dup
include/unistd.h, 200, 定义为函数原型
dup2
include/unistd.h, 248, 定义为函数原型
dupfd
fs/fcntl.c, 18, 定义为函数
E2BIG
include/errno.h, 26, 定义为预处理宏
EACCES
include/errno.h, 32, 定义为预处理宏
EAGAIN
include/errno.h, 30, 定义为预处理宏
EBADF
include/errno.h, 28, 定义为预处理宏
EBUSY
include/errno.h, 35, 定义为预处理宏
ECC_ERR
include/hdreg.h, 49, 定义为预处理宏
ECC_STAT
include/hdreg.h, 26, 定义为预处理宏
ECHILD
include/errno.h, 29, 定义为预处理宏
ECHO
include/termios.h, 172, 定义为预处理宏
ECHOCTL
include/termios.h, 178, 定义为预处理宏
ECHOE
include/termios.h, 173, 定义为预处理宏
ECHOK
include/termios.h, 174, 定义为预处理宏
ECHOKE
include/termios.h, 180, 定义为预处理宏
ECHONL
include/termios.h, 175, 定义为预处理宏
ECHOPRT
include/termios.h, 179, 定义为预处理宏
EDEADLK
include/errno.h, 54, 定义为预处理宏
EDOM
include/errno.h, 52, 定义为预处理宏
EEXIST
include/errno.h, 36, 定义为预处理宏
EFAULT
include/errno.h, 33, 定义为预处理宏
EFBIG
include/errno.h, 46, 定义为预处理宏
EINTR
include/errno.h, 23, 定义为预处理宏
EINVAL
include/errno.h, 41, 定义为预处理宏
EIO
include/errno.h, 24, 定义为预处理宏
EISDIR
include/errno.h, 40, 定义为预处理宏
EMFILE
include/errno.h, 43, 定义为预处理宏
EMLINK
include/errno.h, 50, 定义为预处理宏
EMPTY
include/tty.h, 26, 定义为预处理宏
empty_dir
fs/namei.c, 543, 定义为函数
ENAMETOOLONG
include/errno.h, 55, 定义为预处理宏
end
fs/buffer.c, 29, 定义为变量
end_request
kernel/blk_drv/blk.h, 109, 定义为函数
ENFILE
include/errno.h, 42, 定义为预处理宏
ENODEV
include/errno.h, 38, 定义为预处理宏
ENOENT
include/errno.h, 21, 定义为预处理宏
ENOEXEC
include/errno.h, 27, 定义为预处理宏
ENOLCK
include/errno.h, 56, 定义为预处理宏
ENOMEM
include/errno.h, 31, 定义为预处理宏
ENOSPC
include/errno.h, 47, 定义为预处理宏
ENOSYS

include/errno.h, 57, 定义为预处理宏
 ENOTBLK
 include/errno.h, 34, 定义为预处理宏
 ENOTDIR
 include/errno.h, 39, 定义为预处理宏
 ENOTEMPTY
 include/errno.h, 58, 定义为预处理宏
 ENOTTY
 include/errno.h, 44, 定义为预处理宏
 envp
 init/main.c, 166, 定义为变量
 envp_rc
 init/main.c, 163, 定义为变量
 ENXIO
 include/errno.h, 25, 定义为预处理宏
 EOF_CHAR
 include/tty.h, 40, 定义为预处理宏
 EPERM
 include/errno.h, 20, 定义为预处理宏
 EPIPE
 include/errno.h, 51, 定义为预处理宏
 ERANGE
 include/errno.h, 53, 定义为预处理宏
 ERASE_CHAR
 include/tty.h, 38, 定义为预处理宏
 EROFS
 include/errno.h, 49, 定义为预处理宏
 ERR_STAT
 include/hdreg.h, 24, 定义为预处理宏
 errno
 include/unistd.h, 187, 定义为变量
 include/errno.h, 17, 定义为变量
 lib/errno.c, 7, 定义为变量
 ERROR
 include/errno.h, 19, 定义为预处理宏
 EPIPE
 include/errno.h, 48, 定义为预处理宏
 ESRCH
 include/errno.h, 22, 定义为预处理宏
 ETXTBSY
 include/errno.h, 45, 定义为预处理宏
 EXDEV
 include/errno.h, 37, 定义为预处理宏
 exec
 include/a.out.h, 6, 定义为struct类型
 execl
 include/unistd.h, 204, 定义为函数原型
 execl
 include/unistd.h, 206, 定义为函数原型
 execlp
 include/unistd.h, 205, 定义为函数原型
 execv
 include/unistd.h, 202, 定义为函数原型
 execve
 include/unistd.h, 201, 定义为函数原型
 execvp

include/unistd.h, 203, 定义为函数原型
 exit
 include/unistd.h, 207, 定义为函数原型
 EXT_MEM_K
 init/main.c, 58, 定义为预处理宏
 EXTA
 include/termios.h, 149, 定义为预处理宏
 EXTB
 include/termios.h, 150, 定义为预处理宏
 F_DUPFD
 include/fcntl.h, 23, 定义为预处理宏
 F_GETFD
 include/fcntl.h, 24, 定义为预处理宏
 F_GETFL
 include/fcntl.h, 26, 定义为预处理宏
 F_GETLK
 include/fcntl.h, 28, 定义为预处理宏
 F_OK
 include/unistd.h, 22, 定义为预处理宏
 F_RDLCK
 include/fcntl.h, 38, 定义为预处理宏
 F_SETFD
 include/fcntl.h, 25, 定义为预处理宏
 F_SETFL
 include/fcntl.h, 27, 定义为预处理宏
 F_SETLK
 include/fcntl.h, 29, 定义为预处理宏
 F_SETLKW
 include/fcntl.h, 30, 定义为预处理宏
 F_UNLCK
 include/fcntl.h, 40, 定义为预处理宏
 F_WRLCK
 include/fcntl.h, 39, 定义为预处理宏
 fcntl
 include/unistd.h, 209, 定义为函数原型
 include/fcntl.h, 52, 定义为函数原型
 FD_CLOEXEC
 include/fcntl.h, 33, 定义为预处理宏
 FD_DATA
 include/fdreg.h, 17, 定义为预处理宏
 FD_DCR
 include/fdreg.h, 20, 定义为预处理宏
 FD_DIR
 include/fdreg.h, 19, 定义为预处理宏
 FD_DOR
 include/fdreg.h, 18, 定义为预处理宏
 FD_READ
 include/fdreg.h, 62, 定义为预处理宏
 FD_RECALIBRATE
 include/fdreg.h, 60, 定义为预处理宏
 FD_SEEK
 include/fdreg.h, 61, 定义为预处理宏
 FD_SENSEI
 include/fdreg.h, 64, 定义为预处理宏
 FD_SPECIFY
 include/fdreg.h, 65, 定义为预处理宏

- FD_STATUS
 include/fdreg.h, 16, 定义为预处理宏
 FD_WRITE
 include/fdreg.h, 63, 定义为预处理宏
 FF0
 include/termios.h, 128, 定义为预处理宏
 FF1
 include/termios.h, 129, 定义为预处理宏
 FFDLY
 include/termios.h, 127, 定义为预处理宏
 file
 include/fs.h, 116, 定义为struct类型
 file_read
 fs/read_write.c, 20, 定义为函数原型
 fs/file_dev.c, 17, 定义为函数
 file_table
 fs/file_table.c, 9, 定义为变量
 include/fs.h, 163, 定义为变量
 file_write
 fs/read_write.c, 22, 定义为函数原型
 fs/file_dev.c, 48, 定义为函数
 find_buffer
 fs/buffer.c, 166, 定义为函数
 find_empty_process
 kernel/fork.c, 135, 定义为函数
 find_entry
 fs/namei.c, 91, 定义为函数
 find_first_zero
 fs/bitmap.c, 31, 定义为预处理宏
 FIRST_LDT_ENTRY
 include/sched.h, 154, 定义为预处理宏
 FIRST_TASK
 include/sched.h, 7, 定义为预处理宏
 FIRST_TSS_ENTRY
 include/sched.h, 153, 定义为预处理宏
 flock
 include/fcntl.h, 43, 定义为struct类型
 floppy
 kernel/blk_drv/floppy.c, 114, 定义为变量
 floppy_change
 include/fs.h, 169, 定义为函数原型
 kernel/blk_drv/floppy.c, 139, 定义为函数
 floppy_deselect
 include/fdreg.h, 13, 定义为函数原型
 kernel/blk_drv/floppy.c, 125, 定义为函数
 floppy_init
 init/main.c, 49, 定义为函数原型
 kernel/blk_drv/floppy.c, 457, 定义为函数
 floppy_interrupt
 kernel/blk_drv/floppy.c, 104, 定义为函数原型
 floppy_off
 include/fs.h, 172, 定义为函数原型
 include/fdreg.h, 11, 定义为函数原型
 kernel/sched.c, 240, 定义为函数
 floppy_on
 include/fs.h, 171, 定义为函数原型
 include/fdreg.h, 10, 定义为函数原型
 kernel/sched.c, 232, 定义为函数
 floppy_on_interrupt
 kernel/blk_drv/floppy.c, 404, 定义为函数
 floppy_select
 include/fdreg.h, 12, 定义为函数原型
 floppy_struct
 kernel/blk_drv/floppy.c, 82, 定义为struct类型
 floppy_type
 kernel/blk_drv/floppy.c, 85, 定义为变量
 flush
 kernel/chr_drv/tty_ioctl.c, 39, 定义为函数
 FLUSHO
 include/termios.h, 181, 定义为预处理宏
 fn_ptr
 include/sched.h, 38, 定义为类型
 fork
 include/unistd.h, 210, 定义为函数原型
 free
 include/kernel.h, 12, 定义为预处理宏
 free_block
 fs/bitmap.c, 47, 定义为函数
 include/fs.h, 193, 定义为函数原型
 free_bucket_desc
 lib/malloc.c, 92, 定义为变量
 free_dind
 fs/truncate.c, 29, 定义为函数
 free_ind
 fs/truncate.c, 11, 定义为函数
 free_inode
 fs/bitmap.c, 107, 定义为函数
 include/fs.h, 195, 定义为函数原型
 free_list
 fs/buffer.c, 32, 定义为变量
 free_page
 include/mm.h, 8, 定义为函数原型
 mm/memory.c, 89, 定义为函数
 free_page_tables
 include/sched.h, 30, 定义为函数原型
 mm/memory.c, 105, 定义为函数
 free_s
 include/kernel.h, 10, 定义为函数原型
 lib/malloc.c, 182, 定义为函数
 free_super
 fs/super.c, 40, 定义为函数
 fstat
 include/sys/stat.h, 52, 定义为函数原型
 include/unistd.h, 233, 定义为函数原型
 FULL
 include/tty.h, 29, 定义为预处理宏
 GCC_HEADER
 tools/build.c, 33, 定义为预处理宏
 gdt
 include/head.h, 9, 定义为变量
 GDT_CODE
 include/head.h, 12, 定义为预处理宏

GDT_DATA
include/head.h, 13, 定义为预处理宏
GDT_NUL
include/head.h, 11, 定义为预处理宏
GDT_TMP
include/head.h, 14, 定义为预处理宏
general_protection
kernel/traps.c, 56, 定义为函数原型
get_base
include/sched.h, 226, 定义为预处理宏
get_dir
fs/namei.c, 228, 定义为函数
get_ds
include/asm/segment.h, 54, 定义为函数
get_empty_inode
fs/inode.c, 194, 定义为函数
include/fs.h, 183, 定义为函数原型
get_empty_page
mm/memory.c, 274, 定义为函数
get_free_page
include/mm.h, 6, 定义为函数原型
mm/memory.c, 63, 定义为函数
get_fs
include/asm/segment.h, 47, 定义为函数
get_fs_byte
include/asm/segment.h, 1, 定义为函数
get_fs_long
include/asm/segment.h, 17, 定义为函数
get_fs_word
include/asm/segment.h, 9, 定义为函数
get_hash_table
fs/buffer.c, 183, 定义为函数
include/fs.h, 185, 定义为函数原型
get_limit
include/sched.h, 228, 定义为预处理宏
get_new
kernel/signal.c, 40, 定义为函数
get_pipe_inode
fs/inode.c, 228, 定义为函数
include/fs.h, 184, 定义为函数原型
get_seg_byte
kernel/traps.c, 22, 定义为预处理宏
get_seg_long
kernel/traps.c, 28, 定义为预处理宏
get_super
fs/super.c, 56, 定义为函数
include/fs.h, 197, 定义为函数原型
get_termio
kernel/chr_drv/tty_ioctl.c, 76, 定义为函数
get_termios
kernel/chr_drv/tty_ioctl.c, 56, 定义为函数
getblk
fs/buffer.c, 206, 定义为函数
include/fs.h, 186, 定义为函数原型
GETCH
include/tty.h, 31, 定义为预处理宏

getegid
include/unistd.h, 215, 定义为函数原型
geteuid
include/unistd.h, 213, 定义为函数原型
getgid
include/unistd.h, 214, 定义为函数原型
getpgrp
include/unistd.h, 250, 定义为函数原型
getpid
include/unistd.h, 211, 定义为函数原型
getppid
include/unistd.h, 249, 定义为函数原型
getuid
include/unistd.h, 212, 定义为函数原型
gid_t
include/sys/types.h, 25, 定义为类型
gmtime
include/time.h, 37, 定义为函数原型
gotoxy
kernel/chr_drv/console.c, 88, 定义为函数
hash
fs/buffer.c, 129, 定义为预处理宏
hash_table
fs/buffer.c, 31, 定义为变量
hd
kernel/blk_drv/hd.c, 59, 定义为变量
HD_CMD
include/hdreg.h, 21, 定义为预处理宏
HD_COMMAND
include/hdreg.h, 19, 定义为预处理宏
HD_CURRENT
include/hdreg.h, 16, 定义为预处理宏
HD_DATA
include/hdreg.h, 10, 定义为预处理宏
HD_ERROR
include/hdreg.h, 11, 定义为预处理宏
HD_HCYL
include/hdreg.h, 15, 定义为预处理宏
hd_i_struct
kernel/blk_drv/hd.c, 45, 定义为struct类型
hd_info
kernel/blk_drv/hd.c, 49, 定义为struct类型
kernel/blk_drv/hd.c, 52, 定义为struct类型
hd_init
init/main.c, 48, 定义为函数原型
kernel/blk_drv/hd.c, 343, 定义为函数
hd_interrupt
kernel/blk_drv/hd.c, 67, 定义为函数原型
HD_LCYL
include/hdreg.h, 14, 定义为预处理宏
HD_NSECTOR
include/hdreg.h, 12, 定义为预处理宏
hd_out
kernel/blk_drv/hd.c, 180, 定义为函数
HD_PRECOMP
include/hdreg.h, 18, 定义为预处理宏

- HD_SECTOR
 include/hdreg.h, 13, 定义为预处理宏
 HD_STATUS
 include/hdreg.h, 17, 定义为预处理宏
 hd_struct
 kernel/blk_drv/hd.c, 56, 定义为struct类型
 head
 kernel/blk_drv/floppy.c, 117, 定义为变量
 HIGH_MEMORY
 mm/memory.c, 52, 定义为变量
 HOUR
 kernel/mktime.c, 21, 定义为预处理宏
 HUPCL
 include/termios.h, 160, 定义为预处理宏
 HZ
 include/sched.h, 5, 定义为预处理宏
 I_BLOCK_SPECIAL
 include/const.h, 9, 定义为预处理宏
 I_CHAR_SPECIAL
 include/const.h, 10, 定义为预处理宏
 I_CRNL
 kernel/chr_drv/tty_io.c, 42, 定义为预处理宏
 I_DIRECTORY
 include/const.h, 7, 定义为预处理宏
 I_MAP_SLOTS
 include/fs.h, 39, 定义为预处理宏
 I_NAMED_PIPE
 include/const.h, 11, 定义为预处理宏
 I_NLCR
 kernel/chr_drv/tty_io.c, 41, 定义为预处理宏
 I_NOCR
 kernel/chr_drv/tty_io.c, 43, 定义为预处理宏
 I_REGULAR
 include/const.h, 8, 定义为预处理宏
 I_SET_GID_BIT
 include/const.h, 13, 定义为预处理宏
 I_SET_UID_BIT
 include/const.h, 12, 定义为预处理宏
 I_TYPE
 include/const.h, 6, 定义为预处理宏
 I_UCLC
 kernel/chr_drv/tty_io.c, 40, 定义为预处理宏
 i387_struct
 include/sched.h, 40, 定义为struct类型
 ICANON
 include/termios.h, 170, 定义为预处理宏
 ICRNL
 include/termios.h, 91, 定义为预处理宏
 ID_ERR
 include/hdreg.h, 48, 定义为预处理宏
 idt
 include/head.h, 9, 定义为变量
 IEXTEN
 include/termios.h, 183, 定义为预处理宏
 iget
 fs/inode.c, 244, 定义为函数
 include/fs.h, 182, 定义为函数原型
 IGNBRK
 include/termios.h, 83, 定义为预处理宏
 IGNCR
 include/termios.h, 90, 定义为预处理宏
 IGNPAR
 include/termios.h, 85, 定义为预处理宏
 IMAXBEL
 include/termios.h, 96, 定义为预处理宏
 immouth_p
 kernel/blk_drv/floppy.c, 50, 定义为预处理宏
 IN_ORDER
 kernel/blk_drv/blk.h, 40, 定义为预处理宏
 inb
 include/asm/io.h, 5, 定义为预处理宏
 inb_p
 include/asm/io.h, 17, 定义为预处理宏
 INC
 include/tty.h, 24, 定义为预处理宏
 INC_PIPE
 include/fs.h, 63, 定义为预处理宏
 INDEX_STAT
 include/hdreg.h, 25, 定义为预处理宏
 init
 init/main.c, 45, 定义为函数原型
 init/main.c, 168, 定义为函数
 kernel/chr_drv/serial.c, 26, 定义为函数
 init_bucket_desc
 lib/malloc.c, 97, 定义为函数
 INIT_C_CC
 include/tty.h, 63, 定义为预处理宏
 INIT_REQUEST
 kernel/blk_drv/blk.h, 127, 定义为预处理宏
 INIT_TASK
 include/sched.h, 113, 定义为预处理宏
 init_task
 kernel/sched.c, 58, 定义为union类型
 INLCR
 include/termios.h, 89, 定义为预处理宏
 ino_t
 include/sys/types.h, 27, 定义为类型
 inode_table
 fs/inode.c, 15, 定义为变量
 include/fs.h, 162, 定义为变量
 INODES_PER_BLOCK
 include/fs.h, 55, 定义为预处理宏
 INPCK
 include/termios.h, 87, 定义为预处理宏
 insert_char
 kernel/chr_drv/console.c, 336, 定义为函数
 insert_into_queues
 fs/buffer.c, 149, 定义为函数
 insert_line
 kernel/chr_drv/console.c, 350, 定义为函数
 int3
 kernel/traps.c, 46, 定义为函数原型

- interruptible_sleep_on
include/sched.h, 146, 定义为函数原型
kernel/sched.c, 167, 定义为函数
- INTMASK
kernel/chr_drv/tty_io.c, 19, 定义为预处理宏
- INTR_CHAR
include/tty.h, 36, 定义为预处理宏
- invalid_op
kernel/traps.c, 49, 定义为函数原型
- invalid_TSS
kernel/traps.c, 53, 定义为函数原型
- invalidate
mm/memory.c, 39, 定义为预处理宏
- invalidate_buffers
fs/buffer.c, 84, 定义为函数
- invalidate_inodes
fs/inode.c, 43, 定义为函数
- ioctl
include/unistd.h, 216, 定义为函数原型
- ioctl_ptr
fs/ioctl.c, 15, 定义为类型
- ioctl_table
fs/ioctl.c, 19, 定义为变量
- iput
fs/inode.c, 150, 定义为函数
- include/fs.h, 181, 定义为函数原型
- iret
include/asm/system.h, 20, 定义为预处理宏
- irq13
kernel/traps.c, 61, 定义为函数原型
- is_digit
kernel/vsprintf.c, 16, 定义为预处理宏
- IS_SEEKABLE
include/fs.h, 24, 定义为预处理宏
- isalnum
include/ctype.h, 16, 定义为预处理宏
- isalpha
include/ctype.h, 17, 定义为预处理宏
- isascii
include/ctype.h, 28, 定义为预处理宏
- isctrl
include/ctype.h, 18, 定义为预处理宏
- isdigit
include/ctype.h, 19, 定义为预处理宏
- isgraph
include/ctype.h, 20, 定义为预处理宏
- ISIG
include/termios.h, 169, 定义为预处理宏
- islower
include/ctype.h, 21, 定义为预处理宏
- isprint
include/ctype.h, 22, 定义为预处理宏
- ispunct
include/ctype.h, 23, 定义为预处理宏
- isspace
include/ctype.h, 24, 定义为预处理宏
- ISTRIP
include/termios.h, 88, 定义为预处理宏
- isupper
include/ctype.h, 25, 定义为预处理宏
- isxdigit
include/ctype.h, 26, 定义为预处理宏
- IUCLC
include/termios.h, 92, 定义为预处理宏
- IXANY
include/termios.h, 94, 定义为预处理宏
- IXOFF
include/termios.h, 95, 定义为预处理宏
- IXON
include/termios.h, 93, 定义为预处理宏
- jiffies
include/sched.h, 139, 定义为变量
kernel/sched.c, 60, 定义为变量
- KBD_FINNISH
include/config.h, 19, 定义为预处理宏
- kernel_mktime
init/main.c, 52, 定义为函数原型
kernel/mktime.c, 41, 定义为函数
- keyboard_interrupt
kernel/chr_drv/console.c, 56, 定义为函数原型
- kill
include/unistd.h, 217, 定义为函数原型
include/signal.h, 57, 定义为函数原型
- KILL_CHAR
include/tty.h, 39, 定义为预处理宏
- kill_session
kernel/exit.c, 46, 定义为函数
- KILLMASK
kernel/chr_drv/tty_io.c, 18, 定义为预处理宏
- L_CANON
kernel/chr_drv/tty_io.c, 32, 定义为预处理宏
- L_ECHO
kernel/chr_drv/tty_io.c, 34, 定义为预处理宏
- L_ECHOCTL
kernel/chr_drv/tty_io.c, 37, 定义为预处理宏
- L_ECHOE
kernel/chr_drv/tty_io.c, 35, 定义为预处理宏
- L_ECHOK
kernel/chr_drv/tty_io.c, 36, 定义为预处理宏
- L_ECHOKE
kernel/chr_drv/tty_io.c, 38, 定义为预处理宏
- L_ISIG
kernel/chr_drv/tty_io.c, 33, 定义为预处理宏
- LAST
include/tty.h, 28, 定义为预处理宏
- last_pid
kernel/fork.c, 22, 定义为变量
- LAST_TASK
include/sched.h, 8, 定义为预处理宏
- last_task_used_math
include/sched.h, 137, 定义为变量
kernel/sched.c, 63, 定义为变量

- LATCH
 kernel/sched.c, 46, 定义为预处理宏
 ldiv_t
 include/sys/types.h, 37, 定义为类型
 LDT_CODE
 include/head.h, 17, 定义为预处理宏
 LDT_DATA
 include/head.h, 18, 定义为预处理宏
 LDT_NUL
 include/head.h, 16, 定义为预处理宏
 LEFT
 include/tty.h, 27, 定义为预处理宏
 kernel/vsprintf.c, 31, 定义为预处理宏
 lf
 kernel/chr_drv/console.c, 204, 定义为函数
 link
 include/unistd.h, 218, 定义为函数原型
 ll_rw_block
 include/fs.h, 187, 定义为函数原型
 kernel/blk_drv/ll_rw_blk.c, 145, 定义为函数
 lldt
 include/sched.h, 158, 定义为预处理宏
 localtime
 include/time.h, 38, 定义为函数原型
 lock_buffer
 kernel/blk_drv/ll_rw_blk.c, 42, 定义为函数
 lock_inode
 fs/inode.c, 28, 定义为函数
 lock_super
 fs/super.c, 31, 定义为函数
 LOW_MEM
 mm/memory.c, 43, 定义为预处理宏
 lseek
 include/unistd.h, 219, 定义为函数原型
 ltr
 include/sched.h, 157, 定义为预处理宏
 m_inode
 include/fs.h, 93, 定义为struct类型
 main
 init/main.c, 104, 定义为函数
 tools/build.c, 57, 定义为函数
 main_memory_start
 init/main.c, 100, 定义为变量
 MAJOR
 include/fs.h, 33, 定义为预处理宏
 MAJOR_NR
 kernel/blk_drv/hd.c, 25, 定义为预处理宏
 kernel/blk_drv/floppy.c, 41, 定义为预处理宏
 kernel/blk_drv/ramdisk.c, 17, 定义为预处理宏
 make_request
 kernel/blk_drv/ll_rw_blk.c, 88, 定义为函数
 malloc
 include/kernel.h, 9, 定义为函数原型
 lib/malloc.c, 117, 定义为函数
 MAP_NR
 mm/memory.c, 46, 定义为预处理宏
 MARK_ERR
 include/hdreg.h, 45, 定义为预处理宏
 match
 fs/namei.c, 63, 定义为函数
 math_emulate
 kernel/math/math_emulate.c, 18, 定义为函数
 math_error
 kernel/math/math_emulate.c, 37, 定义为函数
 math_state_restore
 kernel/sched.c, 77, 定义为函数
 MAX
 fs/file_dev.c, 15, 定义为预处理宏
 MAX_ARG_PAGES
 fs/exec.c, 39, 定义为预处理宏
 MAX_ERRORS
 kernel/blk_drv/hd.c, 34, 定义为预处理宏
 kernel/blk_drv/floppy.c, 60, 定义为预处理宏
 MAX_HD
 kernel/blk_drv/hd.c, 35, 定义为预处理宏
 MAX_REPLIES
 kernel/blk_drv/floppy.c, 65, 定义为预处理宏
 MAY_EXEC
 fs/namei.c, 29, 定义为预处理宏
 MAY_READ
 fs/namei.c, 31, 定义为预处理宏
 MAY_WRITE
 fs/namei.c, 30, 定义为预处理宏
 mem_init
 init/main.c, 50, 定义为函数原型
 mm/memory.c, 399, 定义为函数
 mem_map
 mm/memory.c, 57, 定义为变量
 mem_use
 kernel/sched.c, 48, 定义为函数原型
 memchr
 include/string.h, 379, 定义为函数
 memcmp
 include/string.h, 363, 定义为函数
 memcpy
 include/string.h, 336, 定义为函数
 include/asm/memory.h, 8, 定义为预处理宏
 memmove
 include/string.h, 346, 定义为函数
 memory_end
 init/main.c, 98, 定义为变量
 memset
 include/string.h, 395, 定义为函数
 MIN
 fs/file_dev.c, 14, 定义为预处理宏
 MINIX_HEADER
 tools/build.c, 32, 定义为预处理宏
 MINOR
 include/fs.h, 34, 定义为预处理宏
 MINUTE
 kernel/mktime.c, 20, 定义为预处理宏
 mkdir

include/sys/stat.h, 53, 定义为函数原型
mkfifo
include/sys/stat.h, 54, 定义为函数原型
mknod
include/unistd.h, 220, 定义为函数原型
mktime
include/time.h, 33, 定义为函数原型
mode_t
include/sys/types.h, 28, 定义为类型
moff_timer
kernel/sched.c, 203, 定义为变量
mon_timer
kernel/sched.c, 202, 定义为变量
month
kernel/mktime.c, 26, 定义为变量
mount
include/unistd.h, 221, 定义为函数原型
mount_root
fs/super.c, 242, 定义为函数
include/fs.h, 200, 定义为函数原型
move_to_user_mode
include/asm/system.h, 1, 定义为预处理宏
N_ABS
include/a.out.h, 128, 定义为预处理宏
N_BADMAG
include/a.out.h, 31, 定义为预处理宏
N_BSS
include/a.out.h, 137, 定义为预处理宏
N_BSSADDR
include/a.out.h, 107, 定义为预处理宏
N_COMM
include/a.out.h, 140, 定义为预处理宏
N_DATA
include/a.out.h, 134, 定义为预处理宏
N_DATADDR
include/a.out.h, 100, 定义为预处理宏
N_DATOFF
include/a.out.h, 48, 定义为预处理宏
N_DRELOFF
include/a.out.h, 56, 定义为预处理宏
N_EXT
include/a.out.h, 147, 定义为预处理宏
N_FN
include/a.out.h, 143, 定义为预处理宏
N_INDR
include/a.out.h, 164, 定义为预处理宏
N_MAGIC
include/a.out.h, 18, 定义为预处理宏
N_SETA
include/a.out.h, 178, 定义为预处理宏
N_SETB
include/a.out.h, 181, 定义为预处理宏
N_SETD
include/a.out.h, 180, 定义为预处理宏
N_SETT
include/a.out.h, 179, 定义为预处理宏

N_SETV
include/a.out.h, 184, 定义为预处理宏
N_STAB
include/a.out.h, 153, 定义为预处理宏
N_STROFF
include/a.out.h, 64, 定义为预处理宏
N_SYMOFF
include/a.out.h, 60, 定义为预处理宏
N_TEXT
include/a.out.h, 131, 定义为预处理宏
N_TRELOFF
include/a.out.h, 52, 定义为预处理宏
N_TXTADDR
include/a.out.h, 69, 定义为预处理宏
N_TXTOFF
include/a.out.h, 43, 定义为预处理宏
N_TYPE
include/a.out.h, 150, 定义为预处理宏
N_UNDF
include/a.out.h, 125, 定义为预处理宏
NAME_LEN
include/fs.h, 36, 定义为预处理宏
namei
fs/namei.c, 303, 定义为函数
include/fs.h, 178, 定义为函数原型
NCC
include/termios.h, 43, 定义为预处理宏
NCCS
include/termios.h, 53, 定义为预处理宏
new_block
fs/bitmap.c, 75, 定义为函数
include/fs.h, 192, 定义为函数原型
new_inode
fs/bitmap.c, 136, 定义为函数
include/fs.h, 194, 定义为函数原型
next_timer
kernel/sched.c, 270, 定义为变量
nice
include/unistd.h, 222, 定义为函数原型
NL0
include/termios.h, 108, 定义为预处理宏
NL1
include/termios.h, 109, 定义为预处理宏
NLDLY
include/termios.h, 107, 定义为预处理宏
nlink_t
include/sys/types.h, 30, 定义为类型
nlist
include/a.out.h, 111, 定义为struct类型
NMAGIC
include/a.out.h, 25, 定义为预处理宏
nmi
kernel/traps.c, 45, 定义为函数原型
NOFLSH
include/termios.h, 176, 定义为预处理宏
nop

include/asm/system.h, 18, 定义为预处理宏
NPAR
kernel/chr_drv/console.c, 54, 定义为预处理宏
npar
kernel/chr_drv/console.c, 75, 定义为变量
NR_BLK_DEV
kernel/blk_drv/blk.h, 4, 定义为预处理宏
NR_BUFFERS
fs/buffer.c, 34, 定义为变量
include/fs.h, 48, 定义为预处理宏
nr_buffers
include/fs.h, 166, 定义为变量
NR_FILE
include/fs.h, 45, 定义为预处理宏
NR_HASH
include/fs.h, 47, 定义为预处理宏
NR_HD
kernel/blk_drv/hd.c, 50, 定义为预处理宏
kernel/blk_drv/hd.c, 53, 定义为变量
NR_INODE
include/fs.h, 44, 定义为预处理宏
NR_OPEN
include/fs.h, 43, 定义为预处理宏
NR_REQUEST
kernel/blk_drv/blk.h, 15, 定义为预处理宏
NR_SUPER
include/fs.h, 46, 定义为预处理宏
NR_TASKS
include/sched.h, 4, 定义为预处理宏
NRDEVS
fs/char_dev.c, 83, 定义为预处理宏
fs/ioctl.c, 17, 定义为预处理宏
NSIG
include/signal.h, 10, 定义为预处理宏
NULL
include/sys/types.h, 20, 定义为预处理宏
include/unistd.h, 18, 定义为预处理宏
include/stddef.h, 14, 定义为预处理宏
include/stddef.h, 15, 定义为预处理宏
include/string.h, 5, 定义为预处理宏
include/sched.h, 26, 定义为预处理宏
include/fs.h, 52, 定义为预处理宏
number
kernel/vsprintf.c, 40, 定义为函数
O_ACCMODE
include/fcntl.h, 7, 定义为预处理宏
O_APPEND
include/fcntl.h, 15, 定义为预处理宏
O_CREAT
include/fcntl.h, 11, 定义为预处理宏
O_CRNL
kernel/chr_drv/tty_io.c, 47, 定义为预处理宏
O_EXCL
include/fcntl.h, 12, 定义为预处理宏
O_LCUC
kernel/chr_drv/tty_io.c, 49, 定义为预处理宏
O_NDELAY
include/fcntl.h, 17, 定义为预处理宏
O_NLCR
kernel/chr_drv/tty_io.c, 46, 定义为预处理宏
O_NLRET
kernel/chr_drv/tty_io.c, 48, 定义为预处理宏
O_NOCTTY
include/fcntl.h, 13, 定义为预处理宏
O_NONBLOCK
include/fcntl.h, 16, 定义为预处理宏
O_POST
kernel/chr_drv/tty_io.c, 45, 定义为预处理宏
O_RDONLY
include/fcntl.h, 8, 定义为预处理宏
O_RDWR
include/fcntl.h, 10, 定义为预处理宏
O_TRUNC
include/fcntl.h, 14, 定义为预处理宏
O_WRONLY
include/fcntl.h, 9, 定义为预处理宏
OCRNL
include/termios.h, 102, 定义为预处理宏
OFDEL
include/termios.h, 106, 定义为预处理宏
off_t
include/sys/types.h, 32, 定义为类型
offsetof
include/stddef.h, 17, 定义为预处理宏
OFILL
include/termios.h, 105, 定义为预处理宏
OLCUC
include/termios.h, 100, 定义为预处理宏
OMAGIC
include/a.out.h, 23, 定义为预处理宏
ONLCR
include/termios.h, 101, 定义为预处理宏
ONLRET
include/termios.h, 104, 定义为预处理宏
ONOCR
include/termios.h, 103, 定义为预处理宏
oom
mm/memory.c, 33, 定义为函数
open
include/unistd.h, 223, 定义为函数原型
include/fcntl.h, 53, 定义为函数原型
lib/open.c, 11, 定义为函数
open_namei
fs/namei.c, 337, 定义为函数
include/fs.h, 179, 定义为函数原型
OPOST
include/termios.h, 99, 定义为预处理宏
ORIG_ROOT_DEV
init/main.c, 60, 定义为预处理宏
ORIG_VIDEO_COLS
kernel/chr_drv/console.c, 43, 定义为预处理宏
ORIG_VIDEO_EGA_AX

- kernel/chr_drv/console.c, 45, 定义为预处理宏
ORIG_VIDEO_EGA_BX
kernel/chr_drv/console.c, 46, 定义为预处理宏
ORIG_VIDEO_EGA_CX
kernel/chr_drv/console.c, 47, 定义为预处理宏
ORIG_VIDEO_LINES
kernel/chr_drv/console.c, 44, 定义为预处理宏
ORIG_VIDEO_MODE
kernel/chr_drv/console.c, 42, 定义为预处理宏
ORIG_VIDEO_PAGE
kernel/chr_drv/console.c, 41, 定义为预处理宏
ORIG_X
kernel/chr_drv/console.c, 39, 定义为预处理宏
ORIG_Y
kernel/chr_drv/console.c, 40, 定义为预处理宏
origin
kernel/chr_drv/console.c, 69, 定义为变量
outb
include/asm/io.h, 1, 定义为预处理宏
outb_p
include/asm/io.h, 11, 定义为预处理宏
output_byte
kernel/blk_drv/floppy.c, 194, 定义为函数
overflow
kernel/traps.c, 47, 定义为函数原型
PAGE_ALIGN
include/sched.h, 186, 定义为预处理宏
page_exception
kernel/traps.c, 41, 定义为函数原型
page_fault
kernel/traps.c, 57, 定义为函数原型
PAGE_SIZE
include/a.out.h, 79, 定义为预处理宏
include/a.out.h, 88, 定义为预处理宏
include/a.out.h, 92, 定义为预处理宏
include/mm.h, 4, 定义为预处理宏
PAGING_MEMORY
mm/memory.c, 44, 定义为预处理宏
PAGING_PAGES
mm/memory.c, 45, 定义为预处理宏
panic
include/kernel.h, 5, 定义为函数原型
include/sched.h, 35, 定义为函数原型
kernel/panic.c, 16, 定义为函数
par
kernel/chr_drv/console.c, 75, 定义为变量
parallel_interrupt
kernel/traps.c, 60, 定义为函数原型
PARENB
include/termios.h, 165, 定义为预处理宏
PARMRK
include/termios.h, 86, 定义为预处理宏
PARODD
include/termios.h, 166, 定义为预处理宏
partition
include/hdreg.h, 52, 定义为struct类型
pause
include/unistd.h, 224, 定义为函数原型
PENDIN
include/termios.h, 182, 定义为预处理宏
permission
fs/namei.c, 40, 定义为函数
pg_dir
include/head.h, 8, 定义为变量
pid_t
include/sys/types.h, 23, 定义为类型
pipe
include/unistd.h, 225, 定义为函数原型
PIPE_EMPTY
include/fs.h, 61, 定义为预处理宏
PIPE_FULL
include/fs.h, 62, 定义为预处理宏
PIPE_HEAD
include/fs.h, 58, 定义为预处理宏
PIPE_SIZE
include/fs.h, 60, 定义为预处理宏
PIPE_TAIL
include/fs.h, 59, 定义为预处理宏
PLUS
kernel/vsprintf.c, 29, 定义为预处理宏
port_read
kernel/blk_drv/hd.c, 61, 定义为预处理宏
port_write
kernel/blk_drv/hd.c, 64, 定义为预处理宏
pos
kernel/chr_drv/console.c, 71, 定义为变量
printbuf
init/main.c, 42, 定义为变量
printf
include/kernel.h, 6, 定义为函数原型
init/main.c, 151, 定义为函数
printk
include/kernel.h, 7, 定义为函数原型
kernel/printk.c, 21, 定义为函数
ptrdiff_t
include/sys/types.h, 16, 定义为类型
include/stddef.h, 6, 定义为类型
put_fs_byte
include/asm/segment.h, 25, 定义为函数
put_fs_long
include/asm/segment.h, 35, 定义为函数
put_fs_word
include/asm/segment.h, 30, 定义为函数
put_page
include/mm.h, 7, 定义为函数原型
mm/memory.c, 197, 定义为函数
put_super
fs/super.c, 74, 定义为函数
PUTCH
include/tty.h, 33, 定义为预处理宏
ques
kernel/chr_drv/console.c, 76, 定义为变量

- QUIT_CHAR
 include/tty.h, 37, 定义为预处理宏
 QUITMASK
 kernel/chr_drv/tty_io.c, 20, 定义为预处理宏
 quotient
 kernel/chr_drv/tty_ioctl.c, 18, 定义为变量
 R_OK
 include/unistd.h, 25, 定义为预处理宏
 raise
 include/signal.h, 56, 定义为函数原型
 rd_init
 init/main.c, 51, 定义为函数原型
 kernel/blk_drv/ramdisk.c, 52, 定义为函数
 rd_length
 kernel/blk_drv/ramdisk.c, 21, 定义为变量
 rd_load
 kernel/blk_drv/hd.c, 68, 定义为函数原型
 kernel/blk_drv/ramdisk.c, 71, 定义为函数
 rd_start
 kernel/blk_drv/ramdisk.c, 20, 定义为变量
 read
 include/unistd.h, 226, 定义为函数原型
 READ
 include/fs.h, 26, 定义为预处理宏
 read_inode
 fs/inode.c, 17, 定义为函数原型
 fs/inode.c, 294, 定义为函数
 read_intr
 kernel/blk_drv/hd.c, 250, 定义为函数
 read_pipe
 fs/read_write.c, 16, 定义为函数原型
 fs/pipe.c, 13, 定义为函数
 read_super
 fs/super.c, 100, 定义为函数
 READA
 include/fs.h, 28, 定义为预处理宏
 READY_STAT
 include/hdreg.h, 30, 定义为预处理宏
 recal_interrupt
 kernel/blk_drv/floppy.c, 343, 定义为函数
 recal_intr
 kernel/blk_drv/hd.c, 37, 定义为函数原型
 kernel/blk_drv/hd.c, 287, 定义为函数
 recalibrate
 kernel/blk_drv/hd.c, 39, 定义为变量
 kernel/blk_drv/floppy.c, 44, 定义为变量
 recalibrate_floppy
 kernel/blk_drv/floppy.c, 362, 定义为函数
 release
 kernel/exit.c, 19, 定义为函数
 relocation_info
 include/a.out.h, 193, 定义为struct类型
 remove_from_queues
 fs/buffer.c, 131, 定义为函数
 reply_buffer
 kernel/blk_drv/floppy.c, 66, 定义为变量
 request
 kernel/blk_drv/ll_rw_blk.c, 21, 定义为变量
 kernel/blk_drv/blk.h, 23, 定义为struct类型
 kernel/blk_drv/blk.h, 51, 定义为变量
 reserved
 kernel/traps.c, 59, 定义为函数原型
 reset
 kernel/blk_drv/hd.c, 40, 定义为变量
 kernel/blk_drv/floppy.c, 45, 定义为变量
 reset_controller
 kernel/blk_drv/hd.c, 217, 定义为函数
 reset_floppy
 kernel/blk_drv/floppy.c, 386, 定义为函数
 reset_hd
 kernel/blk_drv/hd.c, 230, 定义为函数
 reset_interrupt
 kernel/blk_drv/floppy.c, 373, 定义为函数
 respond
 kernel/chr_drv/console.c, 323, 定义为函数
 RESPONSE
 kernel/chr_drv/console.c, 85, 定义为预处理宏
 restore_cur
 kernel/chr_drv/console.c, 440, 定义为函数
 result
 kernel/blk_drv/floppy.c, 212, 定义为函数
 ri
 kernel/chr_drv/console.c, 214, 定义为函数
 ROOT_DEV
 fs/super.c, 29, 定义为变量
 include/fs.h, 198, 定义为变量
 ROOT_INO
 include/fs.h, 37, 定义为预处理宏
 rs_init
 include/tty.h, 65, 定义为函数原型
 kernel/chr_drv/serial.c, 37, 定义为函数
 rs_write
 include/tty.h, 72, 定义为函数原型
 kernel/chr_drv/serial.c, 53, 定义为函数
 rs1_interrupt
 kernel/chr_drv/serial.c, 23, 定义为函数原型
 rs2_interrupt
 kernel/chr_drv/serial.c, 24, 定义为函数原型
 rw_char
 fs/read_write.c, 15, 定义为函数原型
 fs/char_dev.c, 95, 定义为函数
 rw_interrupt
 kernel/blk_drv/floppy.c, 250, 定义为函数
 rw_kmem
 fs/char_dev.c, 44, 定义为函数
 rw_mem
 fs/char_dev.c, 39, 定义为函数
 rw_memory
 fs/char_dev.c, 65, 定义为函数
 rw_port
 fs/char_dev.c, 49, 定义为函数
 rw_ram

- fs/char_dev.c, 34, 定义为函数
 rw_tty
 fs/char_dev.c, 27, 定义为函数
 rw_ttyx
 fs/char_dev.c, 21, 定义为函数
 S_IFBLK
 include/sys/stat.h, 22, 定义为预处理宏
 S_IFCHR
 include/sys/stat.h, 24, 定义为预处理宏
 S_IFDIR
 include/sys/stat.h, 23, 定义为预处理宏
 S_IFIFO
 include/sys/stat.h, 25, 定义为预处理宏
 S_IFMT
 include/sys/stat.h, 20, 定义为预处理宏
 S_IFREG
 include/sys/stat.h, 21, 定义为预处理宏
 S_IRGRP
 include/sys/stat.h, 42, 定义为预处理宏
 S_IROTH
 include/sys/stat.h, 47, 定义为预处理宏
 S_IRUSR
 include/sys/stat.h, 37, 定义为预处理宏
 S_IRWXG
 include/sys/stat.h, 41, 定义为预处理宏
 S_IRWXO
 include/sys/stat.h, 46, 定义为预处理宏
 S_IRWXU
 include/sys/stat.h, 36, 定义为预处理宏
 S_ISBLK
 include/sys/stat.h, 33, 定义为预处理宏
 S_ISCHR
 include/sys/stat.h, 32, 定义为预处理宏
 S_ISDIR
 include/sys/stat.h, 31, 定义为预处理宏
 S_ISFIFO
 include/sys/stat.h, 34, 定义为预处理宏
 S_ISGID
 include/sys/stat.h, 27, 定义为预处理宏
 S_ISREG
 include/sys/stat.h, 30, 定义为预处理宏
 S_ISUID
 include/sys/stat.h, 26, 定义为预处理宏
 S_ISVTX
 include/sys/stat.h, 28, 定义为预处理宏
 S_IWGRP
 include/sys/stat.h, 43, 定义为预处理宏
 S_IWOTH
 include/sys/stat.h, 48, 定义为预处理宏
 S_IWUSR
 include/sys/stat.h, 38, 定义为预处理宏
 S_IXGRP
 include/sys/stat.h, 44, 定义为预处理宏
 S_IXOTH
 include/sys/stat.h, 49, 定义为预处理宏
 S_IXUSR
 include/sys/stat.h, 39, 定义为预处理宏
 SA_NOCLDSTOP
 include/signal.h, 37, 定义为预处理宏
 SA_NOMASK
 include/signal.h, 38, 定义为预处理宏
 SA_ONESHOT
 include/signal.h, 39, 定义为预处理宏
 save_cur
 kernel/chr_drv/console.c, 434, 定义为函数
 save_old
 kernel/signal.c, 28, 定义为函数
 saved_x
 kernel/chr_drv/console.c, 431, 定义为变量
 saved_y
 kernel/chr_drv/console.c, 432, 定义为变量
 sbrk
 include/unistd.h, 193, 定义为函数原型
 sched_init
 include/sched.h, 32, 定义为函数原型
 kernel/sched.c, 385, 定义为函数
 schedule
 include/sched.h, 33, 定义为函数原型
 kernel/sched.c, 104, 定义为函数
 scr_end
 kernel/chr_drv/console.c, 70, 定义为变量
 scrdown
 kernel/chr_drv/console.c, 170, 定义为函数
 scrup
 kernel/chr_drv/console.c, 107, 定义为函数
 sector
 kernel/blk_drv/floppy.c, 116, 定义为变量
 seek
 kernel/blk_drv/floppy.c, 46, 定义为变量
 SEEK_CUR
 include/unistd.h, 29, 定义为预处理宏
 SEEK_END
 include/unistd.h, 30, 定义为预处理宏
 seek_interrupt
 kernel/blk_drv/floppy.c, 291, 定义为函数
 SEEK_SET
 include/unistd.h, 28, 定义为预处理宏
 SEEK_STAT
 include/hdreg.h, 28, 定义为预处理宏
 seek_track
 kernel/blk_drv/floppy.c, 119, 定义为变量
 segment_not_present
 kernel/traps.c, 54, 定义为函数原型
 SEGMENT_SIZE
 include/a.out.h, 76, 定义为预处理宏
 include/a.out.h, 82, 定义为预处理宏
 include/a.out.h, 85, 定义为预处理宏
 include/a.out.h, 89, 定义为预处理宏
 include/a.out.h, 93, 定义为预处理宏
 selected
 kernel/blk_drv/floppy.c, 122, 定义为变量
 send_break

- kernel/chr_drv/tty_ioctl.c, 51, 定义为函数
 send_sig
 kernel/exit.c, 35, 定义为函数
 set_base
 include/sched.h, 211, 定义为预处理宏
 set_bit
 fs/super.c, 22, 定义为预处理宏
 fs/bitmap.c, 19, 定义为预处理宏
 set_cursor
 kernel/chr_drv/console.c, 313, 定义为函数
 set_fs
 include/asm/segment.h, 61, 定义为函数
 set_intr_gate
 include/asm/system.h, 33, 定义为预处理宏
 set_ldt_desc
 include/asm/system.h, 66, 定义为预处理宏
 set_limit
 include/sched.h, 212, 定义为预处理宏
 set_origin
 kernel/chr_drv/console.c, 97, 定义为函数
 set_system_gate
 include/asm/system.h, 39, 定义为预处理宏
 set_termio
 kernel/chr_drv/tty_ioctl.c, 97, 定义为函数
 set_termios
 kernel/chr_drv/tty_ioctl.c, 66, 定义为函数
 set_trap_gate
 include/asm/system.h, 36, 定义为预处理宏
 set_tss_desc
 include/asm/system.h, 65, 定义为预处理宏
 setgid
 include/unistd.h, 230, 定义为函数原型
 setpgid
 include/unistd.h, 228, 定义为函数原型
 setpgrp
 include/unistd.h, 227, 定义为函数原型
 setsid
 include/unistd.h, 251, 定义为函数原型
 setuid
 include/unistd.h, 229, 定义为函数原型
 setup_DMA
 kernel/blk_drv/floppy.c, 160, 定义为函数
 setup_rw_floppy
 kernel/blk_drv/floppy.c, 269, 定义为函数
 SETUP_SECTS
 tools/build.c, 42, 定义为预处理宏
 share_page
 mm/memory.c, 344, 定义为函数
 show_stat
 kernel/sched.c, 37, 定义为函数
 show_task
 kernel/sched.c, 26, 定义为函数
 sig_atomic_t
 include/signal.h, 6, 定义为类型
 SIG_BLOCK
 include/signal.h, 41, 定义为预处理宏
 SIG_DFL
 include/signal.h, 45, 定义为预处理宏
 SIG_IGN
 include/signal.h, 46, 定义为预处理宏
 SIG_SETMASK
 include/signal.h, 43, 定义为预处理宏
 SIG_UNBLOCK
 include/signal.h, 42, 定义为预处理宏
 SIGABRT
 include/signal.h, 17, 定义为预处理宏
 sigaction
 include/signal.h, 48, 定义为struct类型
 include/signal.h, 66, 定义为函数原型
 sigaddset
 include/signal.h, 58, 定义为函数原型
 SIGALRM
 include/signal.h, 26, 定义为预处理宏
 SIGCHLD
 include/signal.h, 29, 定义为预处理宏
 SIGCONT
 include/signal.h, 30, 定义为预处理宏
 sigdelset
 include/signal.h, 59, 定义为函数原型
 sigemptyset
 include/signal.h, 60, 定义为函数原型
 sigfillset
 include/signal.h, 61, 定义为函数原型
 SIGFPE
 include/signal.h, 20, 定义为预处理宏
 SIGHUP
 include/signal.h, 12, 定义为预处理宏
 SIGILL
 include/signal.h, 15, 定义为预处理宏
 SIGINT
 include/signal.h, 13, 定义为预处理宏
 SIGIOT
 include/signal.h, 18, 定义为预处理宏
 sigismember
 include/signal.h, 62, 定义为函数原型
 SIGKILL
 include/signal.h, 21, 定义为预处理宏
 SIGN
 kernel/vsprintf.c, 28, 定义为预处理宏
 sigpending
 include/signal.h, 63, 定义为函数原型
 SIGPIPE
 include/signal.h, 25, 定义为预处理宏
 sigprocmask
 include/signal.h, 64, 定义为函数原型
 SIGQUIT
 include/signal.h, 14, 定义为预处理宏
 SIGSEGV
 include/signal.h, 23, 定义为预处理宏
 sigset_t
 include/signal.h, 7, 定义为类型
 SIGSTKFLT

- include/signal.h, 28, 定义为预处理宏
SIGSTOP
include/signal.h, 31, 定义为预处理宏
sigsuspend
include/signal.h, 65, 定义为函数原型
SIGTERM
include/signal.h, 27, 定义为预处理宏
SIGTRAP
include/signal.h, 16, 定义为预处理宏
SIGTSTP
include/signal.h, 32, 定义为预处理宏
SIGTTIN
include/signal.h, 33, 定义为预处理宏
SIGTTOU
include/signal.h, 34, 定义为预处理宏
SIGUNUSED
include/signal.h, 19, 定义为预处理宏
SIGUSR1
include/signal.h, 22, 定义为预处理宏
SIGUSR2
include/signal.h, 24, 定义为预处理宏
size_t
include/sys/types.h, 6, 定义为类型
include/time.h, 11, 定义为类型
include/stddef.h, 11, 定义为类型
include/string.h, 10, 定义为类型
skip_atoi
kernel/vsprintf.c, 18, 定义为函数
sleep_if_empty
kernel/chr_drv/tty_io.c, 122, 定义为函数
sleep_if_full
kernel/chr_drv/tty_io.c, 130, 定义为函数
sleep_on
include/sched.h, 145, 定义为函数原型
kernel/sched.c, 151, 定义为函数
SMALL
kernel/vsprintf.c, 33, 定义为预处理宏
SPACE
kernel/vsprintf.c, 30, 定义为预处理宏
SPECIAL
kernel/vsprintf.c, 32, 定义为预处理宏
speed_t
include/termios.h, 214, 定义为类型
ST0
kernel/blk_drv/floppy.c, 67, 定义为预处理宏
ST0_DS
include/fdreg.h, 30, 定义为预处理宏
ST0_ECE
include/fdreg.h, 33, 定义为预处理宏
ST0_HA
include/fdreg.h, 31, 定义为预处理宏
ST0_INTR
include/fdreg.h, 35, 定义为预处理宏
ST0_NR
include/fdreg.h, 32, 定义为预处理宏
ST0_SE
include/fdreg.h, 34, 定义为预处理宏
ST1
kernel/blk_drv/floppy.c, 68, 定义为预处理宏
ST1_CRC
include/fdreg.h, 42, 定义为预处理宏
ST1_EOC
include/fdreg.h, 43, 定义为预处理宏
ST1_MAM
include/fdreg.h, 38, 定义为预处理宏
ST1_ND
include/fdreg.h, 40, 定义为预处理宏
ST1_OR
include/fdreg.h, 41, 定义为预处理宏
ST1_WP
include/fdreg.h, 39, 定义为预处理宏
ST2
kernel/blk_drv/floppy.c, 69, 定义为预处理宏
ST2_BC
include/fdreg.h, 47, 定义为预处理宏
ST2_CM
include/fdreg.h, 52, 定义为预处理宏
ST2_CRC
include/fdreg.h, 51, 定义为预处理宏
ST2_MAM
include/fdreg.h, 46, 定义为预处理宏
ST2_SEH
include/fdreg.h, 49, 定义为预处理宏
ST2_SNS
include/fdreg.h, 48, 定义为预处理宏
ST2_WC
include/fdreg.h, 50, 定义为预处理宏
ST3
kernel/blk_drv/floppy.c, 70, 定义为预处理宏
ST3_HA
include/fdreg.h, 55, 定义为预处理宏
ST3_TZ
include/fdreg.h, 56, 定义为预处理宏
ST3_WP
include/fdreg.h, 57, 定义为预处理宏
stack_segment
kernel/traps.c, 55, 定义为函数原型
start_buffer
fs/buffer.c, 30, 定义为变量
include/fs.h, 165, 定义为变量
START_CHAR
include/tty.h, 41, 定义为预处理宏
startup_time
include/sched.h, 140, 定义为变量
init/main.c, 53, 定义为变量
kernel/sched.c, 61, 定义为变量
stat
include/sys/stat.h, 6, 定义为struct类型
include/sys/stat.h, 55, 定义为函数原型
include/unistd.h, 232, 定义为函数原型
state
kernel/chr_drv/console.c, 74, 定义为变量

STATUS_BUSY
include/fdreg.h, 24, 定义为预处理宏
STATUS_BUSYMASK
include/fdreg.h, 23, 定义为预处理宏
STATUS_DIR
include/fdreg.h, 26, 定义为预处理宏
STATUS_DMA
include/fdreg.h, 25, 定义为预处理宏
STATUS_READY
include/fdreg.h, 27, 定义为预处理宏
STDERR_FILENO
include/unistd.h, 15, 定义为预处理宏
STDIN_FILENO
include/unistd.h, 13, 定义为预处理宏
STDOUT_FILENO
include/unistd.h, 14, 定义为预处理宏
sti
include/asm/system.h, 16, 定义为预处理宏
stime
include/unistd.h, 234, 定义为函数原型
STOP_CHAR
include/tty.h, 42, 定义为预处理宏
str
include/sched.h, 159, 定义为预处理宏
strcat
include/string.h, 54, 定义为函数
strchr
include/string.h, 128, 定义为函数
strcmp
include/string.h, 88, 定义为函数
strcpy
include/string.h, 27, 定义为函数
strcspn
include/string.h, 185, 定义为函数
strerror
include/string.h, 13, 定义为函数原型
strftime
include/time.h, 39, 定义为函数原型
STRINGIFY
tools/build.c, 44, 定义为预处理宏
strlen
include/string.h, 263, 定义为函数
strncat
include/string.h, 68, 定义为函数
strncmp
include/string.h, 107, 定义为函数
strncpy
include/string.h, 38, 定义为函数
strpbrk
include/string.h, 209, 定义为函数
strrchr
include/string.h, 145, 定义为函数
strspn
include/string.h, 161, 定义为函数
strstr
include/string.h, 236, 定义为函数
strtok
include/string.h, 277, 定义为函数
super_block
fs/super.c, 27, 定义为变量
include/fs.h, 124, 定义为struct类型
include/fs.h, 164, 定义为变量
SUPER_MAGIC
include/fs.h, 41, 定义为预处理宏
suser
include/kernel.h, 21, 定义为预处理宏
SUSPEND_CHAR
include/tty.h, 43, 定义为预处理宏
switch_to
include/sched.h, 171, 定义为预处理宏
sync
include/unistd.h, 235, 定义为函数原型
sync_dev
fs/buffer.c, 59, 定义为函数
fs/super.c, 18, 定义为函数原型
include/fs.h, 196, 定义为函数原型
sync_inodes
fs/inode.c, 59, 定义为函数
include/fs.h, 174, 定义为函数原型
sys_access
fs/open.c, 47, 定义为函数
include/sys.h, 34, 定义为函数原型
sys_acct
include/sys.h, 52, 定义为函数原型
kernel/sys.c, 77, 定义为函数
sys_alarm
include/sys.h, 28, 定义为函数原型
kernel/sched.c, 338, 定义为函数
sys_break
include/sys.h, 18, 定义为函数原型
kernel/sys.c, 21, 定义为函数
sys_brk
include/sys.h, 46, 定义为函数原型
kernel/sys.c, 168, 定义为函数
sys_call_table
include/sys.h, 74, 定义为变量
sys_chdir
fs/open.c, 75, 定义为函数
include/sys.h, 13, 定义为函数原型
sys_chmod
fs/open.c, 105, 定义为函数
include/sys.h, 16, 定义为函数原型
sys_chown
fs/open.c, 121, 定义为函数
include/sys.h, 17, 定义为函数原型
sys_chroot
fs/open.c, 90, 定义为函数
include/sys.h, 62, 定义为函数原型
sys_close
fs/open.c, 192, 定义为函数
fs/exec.c, 32, 定义为函数原型
fs/fcntl.c, 16, 定义为函数原型

include/sys.h, 7, 定义为函数原型
kernel/exit.c, 17, 定义为函数原型
sys_creat
fs/open.c, 187, 定义为函数
include/sys.h, 9, 定义为函数原型
sys_dup
fs/fcntl.c, 42, 定义为函数
include/sys.h, 42, 定义为函数原型
sys_dup2
fs/fcntl.c, 36, 定义为函数
include/sys.h, 64, 定义为函数原型
sys_execve
include/sys.h, 12, 定义为函数原型
sys_exit
fs/exec.c, 31, 定义为函数原型
include/sys.h, 2, 定义为函数原型
kernel/exit.c, 137, 定义为函数
sys_fcntl
fs/fcntl.c, 47, 定义为函数
include/sys.h, 56, 定义为函数原型
sys_fork
include/sys.h, 3, 定义为函数原型
sys_fstat
fs/stat.c, 47, 定义为函数
include/sys.h, 29, 定义为函数原型
sys_ftime
include/sys.h, 36, 定义为函数原型
kernel/sys.c, 16, 定义为函数
sys_getegid
include/sys.h, 51, 定义为函数原型
kernel/sched.c, 373, 定义为函数
sys_geteuid
include/sys.h, 50, 定义为函数原型
kernel/sched.c, 363, 定义为函数
sys_getgid
include/sys.h, 48, 定义为函数原型
kernel/sched.c, 368, 定义为函数
sys_getpgrp
include/sys.h, 66, 定义为函数原型
kernel/sys.c, 201, 定义为函数
sys_getpid
include/sys.h, 21, 定义为函数原型
kernel/sched.c, 348, 定义为函数
sys_getppid
include/sys.h, 65, 定义为函数原型
kernel/sched.c, 353, 定义为函数
sys_getuid
include/sys.h, 25, 定义为函数原型
kernel/sched.c, 358, 定义为函数
sys_gtty
include/sys.h, 33, 定义为函数原型
kernel/sys.c, 36, 定义为函数
sys_ioctl
fs/ioctl.c, 30, 定义为函数
include/sys.h, 55, 定义为函数原型

sys_kill
include/sys.h, 38, 定义为函数原型
kernel/exit.c, 60, 定义为函数
sys_link
fs/namei.c, 721, 定义为函数
include/sys.h, 10, 定义为函数原型
sys_lock
include/sys.h, 54, 定义为函数原型
kernel/sys.c, 87, 定义为函数
sys_lseek
fs/read_write.c, 25, 定义为函数
include/sys.h, 20, 定义为函数原型
sys_mkdir
fs/namei.c, 463, 定义为函数
include/sys.h, 40, 定义为函数原型
sys_mknod
fs/namei.c, 412, 定义为函数
include/sys.h, 15, 定义为函数原型
sys_mount
fs/super.c, 200, 定义为函数
include/sys.h, 22, 定义为函数原型
sys_mpx
include/sys.h, 57, 定义为函数原型
kernel/sys.c, 92, 定义为函数
sys_nice
include/sys.h, 35, 定义为函数原型
kernel/sched.c, 378, 定义为函数
sys_open
fs/open.c, 138, 定义为函数
include/sys.h, 6, 定义为函数原型
sys_pause
include/sys.h, 30, 定义为函数原型
kernel/sched.c, 144, 定义为函数
kernel/exit.c, 16, 定义为函数原型
sys_phys
include/sys.h, 53, 定义为函数原型
kernel/sys.c, 82, 定义为函数
sys_pipe
fs/pipe.c, 71, 定义为函数
include/sys.h, 43, 定义为函数原型
sys_prof
include/sys.h, 45, 定义为函数原型
kernel/sys.c, 46, 定义为函数
sys_ptrace
include/sys.h, 27, 定义为函数原型
kernel/sys.c, 26, 定义为函数
sys_read
fs/read_write.c, 55, 定义为函数
include/sys.h, 4, 定义为函数原型
sys_rename
include/sys.h, 39, 定义为函数原型
kernel/sys.c, 41, 定义为函数
sys_rmdir
fs/namei.c, 587, 定义为函数
include/sys.h, 41, 定义为函数原型

- `sys_setgid`
include/sys.h, 47, 定义为函数原型
kernel/sys.c, 72, 定义为函数
- `sys_setpgid`
include/sys.h, 58, 定义为函数原型
kernel/sys.c, 181, 定义为函数
- `sys_setregid`
include/sys.h, 72, 定义为函数原型
kernel/sys.c, 51, 定义为函数
- `sys_setreuid`
include/sys.h, 71, 定义为函数原型
kernel/sys.c, 118, 定义为函数
- `sys_setsid`
include/sys.h, 67, 定义为函数原型
kernel/sys.c, 206, 定义为函数
- `sys_setuid`
include/sys.h, 24, 定义为函数原型
kernel/sys.c, 143, 定义为函数
- `sys_setup`
include/sys.h, 1, 定义为函数原型
kernel/blk_drv/hd.c, 71, 定义为函数
- `sys_sgetmask`
include/sys.h, 69, 定义为函数原型
kernel/signal.c, 15, 定义为函数
- `sys_sigaction`
include/sys.h, 68, 定义为函数原型
kernel/signal.c, 63, 定义为函数
- `sys_signal`
include/sys.h, 49, 定义为函数原型
kernel/signal.c, 48, 定义为函数
- `SYS_SIZE`
tools/build.c, 35, 定义为预处理宏
- `sys_ssetmask`
include/sys.h, 70, 定义为函数原型
kernel/signal.c, 20, 定义为函数
- `sys_stat`
fs/stat.c, 36, 定义为函数
include/sys.h, 19, 定义为函数原型
- `sys_stime`
include/sys.h, 26, 定义为函数原型
kernel/sys.c, 148, 定义为函数
- `sys_stty`
include/sys.h, 32, 定义为函数原型
kernel/sys.c, 31, 定义为函数
- `sys_sync`
fs/buffer.c, 44, 定义为函数
include/sys.h, 37, 定义为函数原型
kernel/panic.c, 14, 定义为函数原型
- `sys_time`
include/sys.h, 14, 定义为函数原型
kernel/sys.c, 102, 定义为函数
- `sys_times`
include/sys.h, 44, 定义为函数原型
kernel/sys.c, 156, 定义为函数
- `sys_ulimit`
include/sys.h, 59, 定义为函数原型
kernel/sys.c, 97, 定义为函数
- `sys_umask`
include/sys.h, 61, 定义为函数原型
kernel/sys.c, 230, 定义为函数
- `sys_umount`
fs/super.c, 167, 定义为函数
include/sys.h, 23, 定义为函数原型
- `sys_uname`
include/sys.h, 60, 定义为函数原型
kernel/sys.c, 216, 定义为函数
- `sys_unlink`
fs/namei.c, 663, 定义为函数
include/sys.h, 11, 定义为函数原型
- `sys_ustat`
fs/open.c, 19, 定义为函数
include/sys.h, 63, 定义为函数原型
- `sys_utime`
fs/open.c, 24, 定义为函数
include/sys.h, 31, 定义为函数原型
- `sys_waitpid`
include/sys.h, 8, 定义为函数原型
kernel/exit.c, 142, 定义为函数
- `sys_write`
fs/read_write.c, 83, 定义为函数
include/sys.h, 5, 定义为函数原型
- `sysbeep`
kernel/chr_drv/console.c, 79, 定义为函数原型
kernel/chr_drv/console.c, 699, 定义为函数
- `sysbeepstop`
kernel/chr_drv/console.c, 691, 定义为函数
- `system_call`
kernel/sched.c, 51, 定义为函数原型
- `TAB0`
include/termios.h, 116, 定义为预处理宏
- `TAB1`
include/termios.h, 117, 定义为预处理宏
- `TAB2`
include/termios.h, 118, 定义为预处理宏
- `TAB3`
include/termios.h, 119, 定义为预处理宏
- `TABDLY`
include/termios.h, 115, 定义为预处理宏
- `table_list`
kernel/chr_drv/tty_io.c, 99, 定义为变量
- `task`
include/sched.h, 136, 定义为变量
kernel/sched.c, 65, 定义为变量
- `TASK_INTERRUPTIBLE`
include/sched.h, 20, 定义为预处理宏
- `TASK_RUNNING`
include/sched.h, 19, 定义为预处理宏
- `TASK_STOPPED`
include/sched.h, 23, 定义为预处理宏
- `task_struct`
include/sched.h, 78, 定义为struct类型

TASK_UNINTERRUPTIBLE
include/sched.h, 21, 定义为预处理宏
task_union
kernel/sched.c, 53, 定义为union类型
TASK_ZOMBIE
include/sched.h, 22, 定义为预处理宏
tcdrain
include/termios.h, 220, 定义为函数原型
tcflow
include/termios.h, 221, 定义为函数原型
TCFLSH
include/termios.h, 18, 定义为预处理宏
tcflush
include/termios.h, 222, 定义为函数原型
TCGETA
include/termios.h, 12, 定义为预处理宏
tcgetattr
include/termios.h, 223, 定义为函数原型
TCGETS
include/termios.h, 8, 定义为预处理宏
TCIFLUSH
include/termios.h, 205, 定义为预处理宏
TCIOFF
include/termios.h, 201, 定义为预处理宏
TCIOFLUSH
include/termios.h, 207, 定义为预处理宏
TCION
include/termios.h, 202, 定义为预处理宏
TCOFLUSH
include/termios.h, 206, 定义为预处理宏
TCOOFF
include/termios.h, 199, 定义为预处理宏
TCOON
include/termios.h, 200, 定义为预处理宏
TCSADRAIN
include/termios.h, 211, 定义为预处理宏
TCSAFLUSH
include/termios.h, 212, 定义为预处理宏
TCSANOW
include/termios.h, 210, 定义为预处理宏
TCSBRK
include/termios.h, 16, 定义为预处理宏
tcsendbreak
include/termios.h, 224, 定义为函数原型
TCSETA
include/termios.h, 13, 定义为预处理宏
TCSETAF
include/termios.h, 15, 定义为预处理宏
tcsetattr
include/termios.h, 225, 定义为函数原型
TCSETAW
include/termios.h, 14, 定义为预处理宏
TCSETS
include/termios.h, 9, 定义为预处理宏
TCSETSF
include/termios.h, 11, 定义为预处理宏
TCSETSW

include/termios.h, 10, 定义为预处理宏
TCXONC
include/termios.h, 17, 定义为预处理宏
tell_father
kernel/exit.c, 83, 定义为函数
termio
include/termios.h, 44, 定义为struct类型
termios
include/termios.h, 54, 定义为struct类型
ticks_to_floppy_on
include/fs.h, 170, 定义为函数原型
include/fdreg.h, 9, 定义为函数原型
kernel/sched.c, 206, 定义为函数
time
include/unistd.h, 236, 定义为函数原型
include/time.h, 31, 定义为函数原型
time_init
init/main.c, 76, 定义为函数
TIME_REQUESTS
kernel/sched.c, 264, 定义为预处理宏
time_t
include/sys/types.h, 11, 定义为类型
include/time.h, 6, 定义为类型
timer_interrupt
kernel/sched.c, 50, 定义为函数原型
timer_list
kernel/sched.c, 266, 定义为struct类型
kernel/sched.c, 270, 定义为变量
times
include/sys/times.h, 13, 定义为函数原型
include/unistd.h, 237, 定义为函数原型
TIOCEXCL
include/termios.h, 19, 定义为预处理宏
TIOCGPGRP
include/termios.h, 22, 定义为预处理宏
TIOCGSOFTCAR
include/termios.h, 32, 定义为预处理宏
TIOCGWINSZ
include/termios.h, 26, 定义为预处理宏
TIOCINQ
include/termios.h, 34, 定义为预处理宏
TIOCM_CAR
include/termios.h, 192, 定义为预处理宏
TIOCM_CD
include/termios.h, 195, 定义为预处理宏
TIOCM_CTS
include/termios.h, 191, 定义为预处理宏
TIOCM_DSR
include/termios.h, 194, 定义为预处理宏
TIOCM_DTR
include/termios.h, 187, 定义为预处理宏
TIOCM_LE
include/termios.h, 186, 定义为预处理宏
TIOCM_RI
include/termios.h, 196, 定义为预处理宏
TIOCM_RNG

- include/termios.h, 193, 定义为预处理宏
TIOCM_RTS
include/termios.h, 188, 定义为预处理宏
TIOCM_SR
include/termios.h, 190, 定义为预处理宏
TIOCM_ST
include/termios.h, 189, 定义为预处理宏
TIOCMBIC
include/termios.h, 30, 定义为预处理宏
TIOCMBIS
include/termios.h, 29, 定义为预处理宏
TIOCMGET
include/termios.h, 28, 定义为预处理宏
TIOCMBSET
include/termios.h, 31, 定义为预处理宏
TIOCNXCL
include/termios.h, 20, 定义为预处理宏
TIOCOUTQ
include/termios.h, 24, 定义为预处理宏
TIOCSCTTY
include/termios.h, 21, 定义为预处理宏
TIOCSPPGRP
include/termios.h, 23, 定义为预处理宏
TIOCSOFTCAR
include/termios.h, 33, 定义为预处理宏
TIOCSTI
include/termios.h, 25, 定义为预处理宏
TIOCSWINSZ
include/termios.h, 27, 定义为预处理宏
tm
include/time.h, 18, 定义为struct类型
tmp_floppy_area
kernel/blk_drv/floppy.c, 105, 定义为变量
tms
include/sys/times.h, 6, 定义为struct类型
toascii
include/ctype.h, 29, 定义为预处理宏
tolower
include/ctype.h, 31, 定义为预处理宏
top
kernel/chr_drv/console.c, 73, 定义为变量
TOSTOP
include/termios.h, 177, 定义为预处理宏
toupper
include/ctype.h, 32, 定义为预处理宏
track
kernel/blk_drv/floppy.c, 118, 定义为变量
transfer
kernel/blk_drv/floppy.c, 309, 定义为函数
trap_init
include/sched.h, 34, 定义为函数原型
kernel/traps.c, 181, 定义为函数
TRK0_ERR
include/hdreg.h, 46, 定义为预处理宏
truncate
fs/truncate.c, 47, 定义为函数
include/fs.h, 173, 定义为函数原型
try_to_share
mm/memory.c, 292, 定义为函数
tss_struct
include/sched.h, 51, 定义为struct类型
TSTPMASK
kernel/chr_drv/tty_io.c, 21, 定义为预处理宏
TTY_BUF_SIZE
include/termios.h, 4, 定义为预处理宏
include/tty.h, 14, 定义为预处理宏
tty_init
include/tty.h, 67, 定义为函数原型
kernel/chr_drv/tty_io.c, 105, 定义为函数
tty_intr
kernel/chr_drv/tty_io.c, 111, 定义为函数
tty_ioctl
fs/ioctl.c, 13, 定义为函数原型
kernel/chr_drv/tty_ioctl.c, 115, 定义为函数
tty_queue
include/tty.h, 16, 定义为struct类型
tty_read
fs/char_dev.c, 16, 定义为函数原型
include/tty.h, 69, 定义为函数原型
kernel/chr_drv/tty_io.c, 230, 定义为函数
tty_struct
include/tty.h, 45, 定义为struct类型
tty_table
include/tty.h, 55, 定义为struct类型
kernel/chr_drv/tty_io.c, 51, 定义为struct类型
tty_write
fs/char_dev.c, 17, 定义为函数原型
include/kernel.h, 8, 定义为函数原型
include/sched.h, 36, 定义为函数原型
include/tty.h, 70, 定义为函数原型
kernel/chr_drv/tty_io.c, 290, 定义为函数
TYPE
kernel/blk_drv/floppy.c, 53, 定义为预处理宏
tzset
include/time.h, 40, 定义为函数原型
u_char
include/sys/types.h, 33, 定义为类型
uid_t
include/sys/types.h, 24, 定义为类型
ulimit
include/unistd.h, 238, 定义为函数原型
umask
include/sys/stat.h, 56, 定义为函数原型
include/unistd.h, 239, 定义为函数原型
umode_t
include/sys/types.h, 29, 定义为类型
umount
include/unistd.h, 240, 定义为函数原型
un_wp_page
mm/memory.c, 221, 定义为函数
uname
include/sys/utsname.h, 14, 定义为函数原型

- include/unistd.h, 241, 定义为函数原型
unexpected_floppy_interrupt
kernel/blk_drv/floppy.c, 353, 定义为函数
unexpected_hd_interrupt
kernel/blk_drv/hd.c, 237, 定义为函数
unlink
include/unistd.h, 242, 定义为函数原型
unlock_buffer
kernel/blk_drv/ll_rw_blk.c, 51, 定义为函数
kernel/blk_drv/blk.h, 101, 定义为函数
unlock_inode
fs/inode.c, 37, 定义为函数
usage
tools/build.c, 52, 定义为函数
USED
mm/memory.c, 47, 定义为预处理宏
user_stack
kernel/sched.c, 67, 定义为变量
ushort
include/sys/types.h, 34, 定义为类型
ustat
include/sys/types.h, 39, 定义为struct类型
include/unistd.h, 243, 定义为函数原型
utimbuf
include/utime.h, 6, 定义为struct类型
utime
include/unistd.h, 244, 定义为函数原型
include/utime.h, 11, 定义为函数原型
utsname
include/sys/utsname.h, 6, 定义为struct类型
va_arg
include/stdarg.h, 24, 定义为预处理宏
va_end
include/stdarg.h, 22, 定义为预处理宏
include/stdarg.h, 21, 定义为函数原型
va_list
include/stdarg.h, 4, 定义为类型
va_start
include/stdarg.h, 13, 定义为预处理宏
include/stdarg.h, 16, 定义为预处理宏
VDISCARD
include/termios.h, 77, 定义为预处理宏
VEOF
include/termios.h, 68, 定义为预处理宏
VEOL
include/termios.h, 75, 定义为预处理宏
VEOL2
include/termios.h, 80, 定义为预处理宏
VERASE
include/termios.h, 66, 定义为预处理宏
verify_area
include/kernel.h, 4, 定义为函数原型
kernel/fork.c, 24, 定义为函数
video_erase_char
kernel/chr_drv/console.c, 67, 定义为变量
video_mem_end
kernel/chr_drv/console.c, 64, 定义为变量
video_mem_start
kernel/chr_drv/console.c, 63, 定义为变量
video_num_columns
kernel/chr_drv/console.c, 59, 定义为变量
video_num_lines
kernel/chr_drv/console.c, 61, 定义为变量
video_page
kernel/chr_drv/console.c, 62, 定义为变量
video_port_reg
kernel/chr_drv/console.c, 65, 定义为变量
video_port_val
kernel/chr_drv/console.c, 66, 定义为变量
video_size_row
kernel/chr_drv/console.c, 60, 定义为变量
video_type
kernel/chr_drv/console.c, 58, 定义为变量
VIDEO_TYPE_CGA
kernel/chr_drv/console.c, 50, 定义为预处理宏
VIDEO_TYPE_EGAC
kernel/chr_drv/console.c, 52, 定义为预处理宏
VIDEO_TYPE_EGAM
kernel/chr_drv/console.c, 51, 定义为预处理宏
VIDEO_TYPE_MDA
kernel/chr_drv/console.c, 49, 定义为预处理宏
VINTR
include/termios.h, 64, 定义为预处理宏
VKILL
include/termios.h, 67, 定义为预处理宏
VLNEXT
include/termios.h, 79, 定义为预处理宏
VMIN
include/termios.h, 70, 定义为预处理宏
VQUIT
include/termios.h, 65, 定义为预处理宏
VREPRINT
include/termios.h, 76, 定义为预处理宏
vsprintf
init/main.c, 44, 定义为函数原型
kernel/printf.c, 19, 定义为函数原型
kernel/vsprintf.c, 92, 定义为函数
VSTART
include/termios.h, 72, 定义为预处理宏
VSTOP
include/termios.h, 73, 定义为预处理宏
VSUSP
include/termios.h, 74, 定义为预处理宏
VSWTC
include/termios.h, 71, 定义为预处理宏
VT0
include/termios.h, 125, 定义为预处理宏
VT1
include/termios.h, 126, 定义为预处理宏
VTDLY
include/termios.h, 124, 定义为预处理宏
VTIME

include/termios.h, 69, 定义为预处理宏
VWERASE
include/termios.h, 78, 定义为预处理宏
W_OK
include/unistd.h, 24, 定义为预处理宏
wait
include/sys/wait.h, 20, 定义为函数原型
include/unistd.h, 246, 定义为函数原型
lib/wait.c, 13, 定义为函数
wait_for_keypress
fs/super.c, 19, 定义为函数原型
kernel/chr_drv/tty_io.c, 140, 定义为函数
wait_for_request
kernel/blk_drv/ll_rw_blk.c, 26, 定义为变量
kernel/blk_drv/blk.h, 52, 定义为变量
wait_motor
kernel/sched.c, 201, 定义为变量
wait_on
include/fs.h, 175, 定义为函数原型
wait_on_buffer
fs/buffer.c, 36, 定义为函数
wait_on_floppy_select
kernel/blk_drv/floppy.c, 123, 定义为变量
wait_on_inode
fs/inode.c, 20, 定义为函数
wait_on_super
fs/super.c, 48, 定义为函数
wait_until_sent
kernel/chr_drv/tty_ioctl.c, 46, 定义为函数
waitpid
include/sys/wait.h, 21, 定义为函数原型
include/unistd.h, 245, 定义为函数原型
wake_up
include/sched.h, 147, 定义为函数原型
kernel/sched.c, 188, 定义为函数
WAKEUP_CHARS
kernel/chr_drv/serial.c, 21, 定义为预处理宏
WEXITSTATUS
include/sys/wait.h, 15, 定义为预处理宏
WIFEXITED
include/sys/wait.h, 13, 定义为预处理宏
WIFSIGNALED
include/sys/wait.h, 18, 定义为预处理宏
WIFSTOPPED
include/sys/wait.h, 14, 定义为预处理宏
WIN_DIAGNOSE
include/hdreg.h, 41, 定义为预处理宏
WIN_FORMAT
include/hdreg.h, 38, 定义为预处理宏
WIN_INIT
include/hdreg.h, 39, 定义为预处理宏
WIN_READ
include/hdreg.h, 35, 定义为预处理宏
WIN_RESTORE
include/hdreg.h, 34, 定义为预处理宏
win_result

kernel/blk_drv/hd.c, 169, 定义为函数
WIN_SEEK
include/hdreg.h, 40, 定义为预处理宏
WIN_SPECIFY
include/hdreg.h, 42, 定义为预处理宏
WIN_VERIFY
include/hdreg.h, 37, 定义为预处理宏
WIN_WRITE
include/hdreg.h, 36, 定义为预处理宏
winsize
include/termios.h, 36, 定义为struct类型
WNOHANG
include/sys/wait.h, 10, 定义为预处理宏
WRERR_STAT
include/hdreg.h, 29, 定义为预处理宏
write
include/unistd.h, 247, 定义为函数原型
WRITE
include/fs.h, 27, 定义为预处理宏
write_inode
fs/inode.c, 18, 定义为函数原型
fs/inode.c, 314, 定义为函数
write_intr
kernel/blk_drv/hd.c, 269, 定义为函数
write_pipe
fs/read_write.c, 17, 定义为函数原型
fs/pipe.c, 41, 定义为函数
write_verify
kernel/fork.c, 20, 定义为函数原型
mm/memory.c, 261, 定义为函数
WRITEA
include/fs.h, 29, 定义为预处理宏
WSTOPSIG
include/sys/wait.h, 17, 定义为预处理宏
WTERMSIG
include/sys/wait.h, 16, 定义为预处理宏
WUNTRACED
include/sys/wait.h, 11, 定义为预处理宏
X_OK
include/unistd.h, 23, 定义为预处理宏
XCASE
include/termios.h, 171, 定义为预处理宏
XTABS
include/termios.h, 120, 定义为预处理宏
y
kernel/chr_drv/console.c, 72, 定义为变量
YEAR
kernel/mktime.c, 23, 定义为预处理宏
Z_MAP_SLOTS
include/fs.h, 40, 定义为预处理宏
ZEROPAD
kernel/vsprintf.c, 27, 定义为预处理宏
ZMAGIC
include/a.out.h, 27, 定义为预处理宏

