```
/*
 *  linux/mm/memory.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * demand-loading started 01.12.91 - seems it is high on the list of
 * things wanted, and it should be easy to implement. - Linus
 */
/*
 * 需求加载是从 91.12.1 开始编写的 - 在程序编制表中似乎是最重要的程序，
 * 并且应该是很容易编制的 - Linus
 */

/*
 * Ok, demand-loading was easy, shared pages a little bit tricker. Shared
 * pages started 02.12.91, seems to work. - Linus.
 *
 * Tested sharing by executing about 30 /bin/sh: under the old kernel it
 * would have taken more than the 6M I have free, but it worked well as
 * far as I could see.
 *
 * Also corrected some "invalidate()"s - I wasn't doing enough of them.
 */
/*
 * OK，需求加载是比较容易编写的，而共享页面却需要有点技巧。共享页面程序是
 * 91.12.2 开始编写的，好象能够工作 - Linus。
 *
 * 通过执行大约 30 个/bin/sh 对共享操作进行了测试：在老内核当中需要占用多于
 * 6M 的内存，而目前却不用。现在看来工作得很好。
 *
 * 对"invalidate()"函数也进行了修正 - 在这方面我还做的不够。
 */

/*
 * Real VM (paging to/from disk) started 18.12.91. Much more work and
 * thought has to go into this. Oh, well..
 * 19.12.91  -  works, somewhat. Sometimes I get faults, don't know why.
 *              Found it. Everything seems to work now.
 * 20.12.91  -  Ok, making the swap-device changeable like the root.
 */
/*
 * 91.12.18 开始编写真正的虚拟内存管理 VM（交换页面到/从磁盘）。需要对此
 * 考虑很多并且需要作很多工作。呵呵，也只能这样了。
 * 91.12.19  -  在某种程度上可以工作了，但有时会出错，不知道怎么回事。
 *              找到错误了，现在好像一切都能工作了。
 * 91.12.20  -  OK，把交换设备修改成可更改的了，就像根文件设备那样。
 */

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

```

```
33 #include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等嵌入汇编宏。
34
35 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 的数据。
36 #include <linux/head.h>   // head 头文件，定义段描述符的简单结构，和几个选择符常量。
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
   // CODE_SPACE(addr) ((((addr)+0xfff)&~0xfff)<current->start_code+current->end_code)。
   // 该宏用于判断给定线性地址是否位于当前进程的代码段中，“(((addr)+4095)&~4095)”用于
   // 取得线性地址 addr 所在内存页面的末端地址。参见 265 行。
39 #define CODE_SPACE(addr) ((((addr)+4095)&~4095) < \
40 current->start_code + current->end_code)
41
42 unsigned long HIGH_MEMORY = 0;          // 全局变量，存放实际物理内存最高端地址。
43
   // 从 from 处复制 1 页内存到 to 处（4K 字节）。
44 #define copy_page(from,to) \
45 __asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si")
46
   // 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）。
47 unsigned char mem_map [ PAGING_PAGES ] = {0,};
48
49 /*
50  * Free a page of memory at physical address 'addr'. Used by
51  * 'free_page_tables()'
52  */
   /*
    * 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。
53 void free_page(unsigned long addr)
54 {
   // 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。
55         if (addr < LOW_MEM) return;
56         if (addr >= HIGH_MEMORY)
57                 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。
58         addr -= LOW_MEM;
59         addr >>= 12;
60         if (mem_map[addr]--) return;
61         mem_map[addr]=0;
62         panic("trying to free free page");
63 }
64
```

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

   // 此时 size 是释放的页表个数，即页目录项数，而 dir 是起始目录项指针。现在开始循环
   // 操作页目录项，依次释放每个页表中的页表项。如果当前目录项无效（P 位=0），表示该
   // 目录项没有使用（对应的页表不存在），则继续处理下一个目录项。否则从目录项中取出
   // 页表地址 pg_table，并对该页表中的 1024 个表项进行处理，释放有效页表项（P 位=1）
   // 对应的物理内存页面，或者从交换设备中释放无效页表项（P 位=0）对应的页面，即释放
   // 交换设备中对应的内存页面（因为页面可能已经交换出去）。然后把该页表项清零，并继
   // 续处理下一页表项。当一个页表所有表项都处理完毕就释放该页表自身占据的内存页面，
   // 并继续处理下一页目录项。最后刷新页变换高速缓冲，并返回 0。
80         for ( ; size-->0 ; dir++) {
81                 if (!(1 & *dir))
82                         continue;
83                 pg_table = (unsigned long *) (0xfffff000 & *dir);  // 取页表地址。
84                 for (nr=0 ; nr<1024 ; nr++) {
85                         if (*pg_table) {              // 若所指页表项内容不为 0，则
86                                 if (1 & *pg_table)    // 若该项有效，则释放对应页。
87                                         free_page(0xfffff000 & *pg_table);
88                                 else                  // 否则释放交换设备中对应页。
```

```
89                                      swap_free(*pg_table >> 1);
90                                  *pg_table = 0;          // 该页表项内容清零。
91                              }
92                              pg_table++;                 // 指向页表中下一项。
93                      }
94                      free_page(0xfffff000 & *dir);       // 释放该页表所占内存页面。
95                      *dir = 0;                           // 对应页表的目录项清零。
96              }
97              invalidate();                               // 刷新 CPU 页变换高速缓冲。
98              return 0;
99      }
100
101     /*
102      *  Well, here is one of the most complicated functions in mm. It
103      * copies a range of linerar addresses by copying only the pages.
104      * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
105      *
106      * Note! We don't copy just any chunks of memory - addresses have to
107      * be divisible by 4Mb (one page-directory entry), as this makes the
108      * function easier. It's used only by fork anyway.
109      *
110      * NOTE 2!! When from==0 we are copying kernel space for the first
111      * fork(). Then we DONT want to copy a full page-directory entry, as
112      * that would lead to some serious memory waste - we just copy the
113      * first 160 pages - 640kB. Even that is more than we need, but it
114      * doesn't take any more memory - we don't copy-on-write in the low
115      * 1 Mb-range, so the pages can be shared with the kernel. Thus the
116      * special case for nr=xxxx.
117      */
        /*
         * 好了，下面是内存管理 mm 中最为复杂的程序之一。它通过只复制内存页面
         * 来拷贝一定范围内线性地址中的内容。 希望代码中没有错误，因为我不想
         * 再调试这块代码了:-)。
         *
         * 注意！我们并不复制任何内存块 - 内存块的地址需要是 4Mb 的倍数（正好
         * 一个页目录项对应的内存长度），因为这样处理可使函数很简单。 不管怎
         * 样，它仅被 fork()使用。
         *
         * 注意 2！！ 当 from==0 时，说明是在为第一次 fork()调用复制内核空间。
         * 此时我们就不想复制整个页目录项对应的内存，因为这样做会导致内存严
         * 重浪费 - 我们只须复制开头 160 个页面 - 对应 640kB。即使是复制这些
         * 页面也已经超出我们的需求，但这不会占用更多的内存 - 在低 1Mb 内存
         * 范围内我们不执行写时复制操作，所以这些页面可以与内核共享。因此这
         * 是 nr=xxxx 的特殊情况（nr 在程序中指页面数）。
         */
        //// 复制页目录表项和页表项。
        // 复制指定线性地址和长度内存对应的页目录项和页表项，从而被复制的页目录和页表对应
        // 的原物理内存页面区被两套页表映射而共享使用。复制时，需申请新页面来存放新页表，
        // 原物理内存区将被共享。此后两个进程（父进程和其子进程）将共享内存区，直到有一个
        // 进程执行写操作时，内核才会为写操作进程分配新的内存页（写时复制机制）。
        // 参数 from、to 是线性地址，size 是需要复制（共享）的内存长度，单位是字节。
118     int copy_page_tables(unsigned long from,unsigned long to,long size)
119     {
```

```
120         unsigned long * from_page_table;
121         unsigned long * to_page_table;
122         unsigned long this_page;
123         unsigned long * from_dir, * to_dir;
124         unsigned long new_page;
125         unsigned long nr;
126
    // 首先检测参数给出的源地址 from 和目的地址 to 的有效性。源地址和目的地址都需要在 4Mb
    // 内存边界地址上。否则出错死机。作这样的要求是因为一个页表的 1024 项可管理 4Mb 内存。
    // 源地址 from 和目的地址 to 只有满足这个要求才能保证从一个页表的第 1 项开始复制页表
    // 项，并且新页表的最初所有项都是有效的。  然后取得源地址和目的地址的起始目录项指针
    // （from_dir 和 to_dir）。再根据参数给出的长度 size 计算要复制的内存块占用的页表数
    // （即目录项数）。参见前面对 78、79 行的解释。
127         if ((from&0x3fffff) || (to&0x3fffff))
128                 panic("copy_page_tables called with wrong alignment");
129         from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
130         to_dir = (unsigned long *) ((to>>20) & 0xffc);
131         size = ((unsigned) (size+0x3fffff)) >> 22;
    // 在得到了源起始目录项指针 from_dir 和目的起始目录项指针 to_dir 以及需要复制的页表
    // 个数 size 后，下面开始对每个页目录项依次申请 1 页内存来保存对应的页表，并且开始
    // 页表项复制操作。如果目的目录项指定的页表已经存在（P=1），则出错死机。 如果源目
    // 录项无效，即指定的页表不存在（P=0），则继续循环处理下一个页目录项。
132         for( ; size-->0 ; from_dir++,to_dir++) {
133                 if (1 & *to_dir)
134                         panic("copy_page_tables: already exist");
135                 if (!(1 & *from_dir))
136                         continue;

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

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

    // 此时对于当前页表，开始循环复制指定的 nr 个内存页面表项。先取出源页表项内容，如果
    // 当前源页面没有使用（项内容为 0），则不用复制该表项，继续处理下一项。
142                 for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
143                         this_page = *from_page_table;
144                         if (!this_page)
145                                 continue;
    // 如果该表项有内容，但是其存在位 P=0，则该表项对应的页面可能在交换设备中。于是先申
```

```
    // 请 1 页内存，并从交换设备中读入该页面（若交换设备中有的话）。然后将该页表项复制到
    // 目的页表项中。并修改源页表项内容指向该新申请的内存页，并设置表项标志为“页面脏”
    // 加上 7。然后继续处理下一页表项。否则复位页表项中 R/W 标志（位 1 置 0），即让页表项
    // 对应的内存页面只读，然后将该页表项复制到目的页表中。
146                         if (!(1 & this_page)) {
147                                 if (!(new_page = get_free_page()))
148                                         return -1;
149                                 read_swap_page(this_page>>1, (char *) new_page);
150                                 *to_page_table = this_page;
151                                 *from_page_table = new_page | (PAGE_DIRTY | 7);
152                                 continue;
153                         }
154                         this_page &= ~2;
155                         *to_page_table = this_page;

    // 如果该页表项所指物理页面的地址在 1MB 以上，则需要设置内存页面映射数组 mem_map[]，
    // 于是计算页面号，并以它为索引在页面映射数组相应项中增加引用次数。而对于位于 1MB
    // 以下的页面，说明是内核页面，因此不需要对 mem_map[]进行设置。因为 mem_map[]仅用
    // 于管理主内存区中的页面使用情况。 因此对于内核移动到任务 0 中并且调用 fork()创建
    // 任务 1 时（用于运行 init()），由于此时复制的页面还仍然都在内核代码区域，因此以下
    // 判断中的语句不会执行，任务 0 的页面仍然可以随时读写。只有当调用 fork() 的父进程
    // 代码处于主内存区（页面位置大于 1MB）时才会执行。这种情况需要在进程调用 execve()，
    // 并装载执行了新程序代码时才会出现。
    // 157 行语句含义是令源页表项所指内存页也为只读。因为现在开始已有两个进程共用内存
    // 区了。若其中 1 个进程需要进行写操作，则可以通过页异常写保护处理为执行写操作的进
    // 程分配 1 页新空闲页面，也即进行写时复制（copy on write）操作。
156                         if (this_page > LOW_MEM) {
157                                 *from_page_table = this_page; // 令源页表项也只读。
158                                 this_page -= LOW_MEM;
159                                 this_page >>= 12;
160                                 mem_map[this_page]++;
161                         }
162                 }
163         }
164         invalidate();                           // 刷新页变换高速缓冲。
165         return 0;
166 }
167
168 /*
169  * This function puts a page in memory at the wanted address.
170  * It returns the physical address of the page gotten, 0 if
171  * out of memory (either when trying to access page-table or
172  * page.)
173  */
    /*
     * 下面函数将一内存页面放置（映射）到指定线性地址处。它返回页面
     * 的物理地址，如果内存不够(在访问页表或页面时)，则返回 0。
     */
    // 把一物理内存页面映射到线性地址空间指定处。
    // 或者说是把线性地址空间中指定地址 address 处的页面映射到主内存区页面 page 上。主要
    // 工作是在相关页目录项和页表项中设置指定页面的信息。若成功则返回物理页面地址。 在
    // 处理缺页异常的 C 函数 do_no_page() 中会调用此函数。对于缺页引起的异常，由于任何缺
    // 页缘故而对页表作修改时，并不需要刷新 CPU 的页变换缓冲（或称 Translation Lookaside
```

```
    // Buffer - TLB），即使页表项中标志 P 被从 0 修改成 1。因为无效页项不会被缓冲，因此当
    // 修改了一个无效的页表项时不需要刷新。在此就表现为不用调用 Invalidate()函数。
    // 参数 page 是分配的主内存区中某一页面（页帧，页框）的指针；address 是线性地址。
174 static unsigned long put_page(unsigned long page,unsigned long address)
175 {
176         unsigned long tmp, *page_table;
177
178 /* NOTE !!! This uses the fact that _pg_dir=0 */
    /* 注意!!! 这里使用了页目录表基地址_pg_dir=0 的条件 */
179
    // 首先判断参数给定物理内存页面 page 的有效性。如果该页面位置低于 LOW_MEM（1MB）或
    // 超出系统实际含有内存高端 HIGH_MEMORY，则发出警告。LOW_MEM 是主内存区可能有的最
    // 小起始位置。当系统物理内存小于或等于 6MB 时，主内存区起始于 LOW_MEM 处。再查看一
    // 下该 page 页面是否是已经申请的页面，即判断其在内存页面映射字节图 mem_map[]中相
    // 应字节是否已经置位。若没有则需发出警告。
180         if (page < LOW_MEM || page >= HIGH_MEMORY)
181                 printk("Trying to put page %p at %p\n",page,address);
182         if (mem_map[(page-LOW_MEM)>>12] != 1)
183                 printk("mem_map disagrees with %p at %p\n",page,address);

    // 然后根据参数指定的线性地址 address 计算其在页目录表中对应的目录项指针，并从中取得
    // 二级页表地址。 如果该目录项有效（P=1），即指定的页表在内存中，则从中取得指定页表
    // 地址放到 page_table 变量中。否则申请一空闲页面给页表使用，并在对应目录项中置相应
    // 标志（7 - User、U/S、R/W）。然后将该页表地址放到 page_table 变量中。
184         page_table = (unsigned long *) ((address>>20) & 0xffc);
185         if ((*page_table)&1)
186                 page_table = (unsigned long *) (0xfffff000 & *page_table);
187         else {
188                 if (!(tmp=get_free_page()))
189                         return 0;
190                 *page_table = tmp | 7;
191                 page_table = (unsigned long *) tmp;
192         }
    // 最后在找到的页表 page_table 中设置相关页表项内容，即把物理页面 page 的地址填入表
    // 项同时置位 3 个标志（U/S、W/R、P）。该页表项在页表中的索引值等于线性地址位 21 --
    // 位 12 组成的 10 比特的值。每个页表共可有 1024 项（0 -- 0x3ff）。
193         page_table[(address>>12) & 0x3ff] = page | 7;
194 /* no need for invalidate */
    /* 不需要刷新页变换高速缓冲 */
195         return page;                    // 返回物理页面地址。
196 }
197
198 /*
199  * The previous function doesn't work very well if you also want to mark
200  * the page dirty: exec.c wants this, as it has earlier changed the page,
201  * and we want the dirty-status to be correct (for VM). Thus the same
202  * routine, but this time we mark it dirty too.
203  */
    /*
     * 如果你也想设置页面已修改标志，则上一个函数工作得不是很好：exec.c 程序
     * 需要这种设置。因为 exec.c 中函数会在放置页面之前修改过页面内容。为了实
     * 现 VM，我们需要能正确设置已修改状态标志。因而下面就有了与上面相同的函
     * 数，但是该函数在放置页面时会把页面标志为已修改状态。
```

```
 */
// 把一内容已修改过的物理内存页面映射到线性地址空间指定处。
// 该函数与上一个函数 put_page()几乎完全一样，除了本函数在 223 行设置页表项内容时，
// 同时还设置了页面已修改标志（位 6，PAGE_DIRTY）。
204 unsigned long put_dirty_page(unsigned long page, unsigned long address)
205 {
206         unsigned long tmp, *page_table;
207
208 /* NOTE !!! This uses the fact that _pg_dir=0 */
209
210         if (page < LOW_MEM || page >= HIGH_MEMORY)
211                 printk("Trying to put page %p at %p\n",page,address);
212         if (mem_map[(page-LOW_MEM)>>12] != 1)
213                 printk("mem_map disagrees with %p at %p\n",page,address);
214         page_table = (unsigned long *) ((address>>20) & 0xffc);
215         if ((*page_table)&1)
216                 page_table = (unsigned long *) (0xfffff000 & *page_table);
217         else {
218                 if (!(tmp=get_free_page()))
219                         return 0;
220                 *page_table = tmp|7;
221                 page_table = (unsigned long *) tmp;
222         }
223         page_table[(address>>12) & 0x3ff] = page | (PAGE_DIRTY | 7);
224 /* no need for invalidate */
225         return page;
226 }
227
//// 取消写保护页面函数。用于页异常中断过程中写保护异常的处理（写时复制）。
// 在内核创建进程时，新进程与父进程被设置成共享代码和数据内存页面，并且所有这些页面
// 均被设置成只读页面。而当新进程或原进程需要向内存页面写数据时，CPU 就会检测到这个
// 情况并产生页面写保护异常。于是在这个函数中内核就会首先判断要写的页面是否被共享。
// 若没有则把页面设置成可写然后退出。若页面是出于共享状态，则需要重新申请一新页面并
// 复制被写页面内容，以供写进程单独使用。共享被取消。
// 输入参数为页表项指针，是物理地址。[ un_wp_page -- Un-Write Protect Page]
228 void un_wp_page(unsigned long * table_entry)
229 {
230         unsigned long old_page,new_page;
231
// 首先取参数指定的页表项中物理页面位置（地址）并判断该页面是否是共享页面。如果原
// 页面地址大于内存低端 LOW_MEM（表示在主内存区中），并且其在页面映射字节图数组中
// 值为 1（表示页面仅被引用 1 次，页面没有被共享），则在该页面的页表项中置 R/W 标志
// （可写），并刷新页变换高速缓冲，然后返回。即如果该内存页面此时只被一个进程使用，
// 并且不是内核中的进程，就直接把属性改为可写即可，不用再重新申请一个新页面。
232         old_page = 0xfffff000 & *table_entry;       // 取指定页表项中物理页面地址。
233         if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
234                 *table_entry |= 2;
235                 invalidate();
236                 return;
237         }
// 否则就需要在主内存区内申请一页空闲页面给执行写操作的进程单独使用，取消页面共享。
// 如果原页面大于内存低端（则意味着 mem_map[] > 1，页面是共享的），则将原页面的页
// 面映射字节数组值递减 1。然后将指定页表项内容更新为新页面地址，并置可读写等标志
```

```
    // （U/S、R/W、P）。在刷新页变换高速缓冲之后，最后将原页面内容复制到新页面。
238         if (!(new_page=get_free_page()))
239                 oom();                                  // Out of Memory。内存不够处理。
240         if (old_page >= LOW_MEM)
241                 mem_map[MAP_NR(old_page)]--;
242         copy_page(old_page,new_page);
243         *table_entry = new_page | 7;
244         invalidate();
245 }
246
247 /*
248  * This routine handles present pages, when users try to write
249  * to a shared page. It is done by copying the page to a new address
250  * and decrementing the shared-page counter for the old page.
251  *
252  * If it's in code space we exit with a segment error.
253  */
    /*
     * 当用户试图往一共享页面上写时，该函数处理已存在的内存页面（写时复制），
     * 它是通过将页面复制到一个新地址上并且递减原页面的共享计数值实现的。
     *
     * 如果它在代码空间，我们就显示段出错信息并退出。
     */
    //// 执行写保护页面处理。
    // 是写共享页面处理函数。是页异常中断处理过程中调用的 C 函数。在 page.s 程序中被调用。
    // 函数参数 error_code 和 address 是进程在写写保护页面时由 CPU 产生异常而自动生成的。
    // error_code 指出出错类型，参见本章开始处的“内存页面出错异常”一节；address 是产生
    // 异常的页面线性地址。写共享页面时需复制页面（写时复制）。
254 void do_wp_page(unsigned long error_code,unsigned long address)
255 {
    // 首先判断 CPU 控制寄存器 CR2 给出的引起页面异常的线性地址在什么范围中。如果 address
    // 小于 TASK_SIZE（0x4000000，即 64MB），表示异常页面位置在内核或任务 0 和任务 1 所处
    // 的线性地址范围内，于是发出警告信息“内核范围内存被写保护”；如果 (address - 当前
    // 进程代码起始地址)大于一个进程的长度（64MB），表示 address 所指的线性地址不在引起
    // 异常的进程线性地址空间范围内，则在发出出错信息后退出。
256         if (address < TASK_SIZE)
257                 printk("\n\rBAD! KERNEL MEMORY WP-ERR!\n\r");
258         if (address - current->start_code > TASK_SIZE) {
259                 printk("Bad things happen: page error in do_wp_page\n\r");
260                 do_exit(SIGSEGV);
261         }
262 #if 0
263 /* we cannot do this yet: the estdio library writes to code space */
264 /* stupid, stupid. I really want the libc.a from GNU */
    /* 我们现在还不能这样做：因为 estdio 库会在代码空间执行写操作 */
    /* 真是太愚蠢了。我真想从 GNU 得到 libc.a 库。*/
    // 如果线性地址位于进程的代码空间中，则终止执行程序。因为代码是只读的。
265         if (CODE_SPACE(address))
266                 do_exit(SIGSEGV);
267 #endif
    // 调用上面函数 un_wp_page()来处理取消页面保护。但首先需要为其准备好参数。参数是
    // 线性地址 address 指定页面在页表中的页表项指针，其计算方法是：
    // ① ((address>>10) & 0xffc)：计算指定线性地址中页表项在页表中的偏移地址；因为
```

```
    // 根据线性地址结构，(address>>12) 就是页表项中的索引，但每项占 4 个字节，因此乘
    // 4 后：(address>>12)<<2 = (address>>10)&0xffc 就可得到页表项在表中的偏移地址。
    // 与操作&0xffc 用于限制地址范围在一个页面内。 又因为只移动了 10 位，因此最后 2 位
    // 是线性地址低 12 位中的最高 2 位，也应屏蔽掉。 因此求线性地址中页表项在页表中偏
    // 移地址直观一些的表示方法是(((address>>12) & 0x3ff)<<2 )。
    // ② (0xfffff000 & *((address>>20) &0xffc))：用于取目录项中页表的地址值；其中，
    // ((address>>20) &0xffc)用于取线性地址中的目录索引项在目录表中的偏移位置。因为
    // address>>22 是目录项索引值，但每项 4 个字节，因此乘以 4 后： (address>>22)<<2
    // = (address>>20) 就是指定项在目录表中的偏移地址。 &0xffc 用于屏蔽目录项索引值
    // 中最后 2 位。因为只移动了 20 位，因此最后 2 位是页表索引的内容，应该屏蔽掉。而
    // *((address>>20) &0xffc) 则是取指定目录表项内容中对应页表的物理地址。最后与上
    // 0xfffff000 用于屏蔽掉页目录项内容中的一些标志位（目录项低 12 位）。直观表示为
    // (0xfffff000 & *((unsigned long *) (((address>>22) & 0x3ff)<<2)))。
    // ③ 由①中页表项在页表中偏移地址加上 ②中目录表项内容中对应页表的物理地址即可
    // 得到页表项的指针（物理地址）。这里对共享的页面进行复制。
268         un_wp_page((unsigned long *)
269                 (((address>>10) & 0xffc) + (0xfffff000 &
270                 *((unsigned long *) ((address>>20) &0xffc)))));
271
272 }
273
    //// 写页面验证。
    // 若页面不可写，则复制页面。在 fork.c 中第 34 行被内存验证通用函数 verify_area()调用。
    // 参数 address 是指定页面在 4G 空间中的线性地址。
274 void write_verify(unsigned long address)
275 {
276         unsigned long page;
277
    // 首先取指定线性地址对应的页目录项，根据目录项中的存在位（P）判断目录项对应的页表
    // 是否存在（存在位 P=1?），若不存在（P=0）则返回。这样处理是因为对于不存在的页面没
    // 有共享和写时复制可言，并且若程序对此不存在的页面执行写操作时，系统就会因为缺页异
    // 常而去执行 do_no_page()，并为这个地方使用 put_page()函数映射一个物理页面。
    // 接着程序从目录项中取页表地址，加上指定页面在页表中的页表项偏移值，得对应地址的页
    // 表项指针。在该表项中包含着给定线性地址对应的物理页面。
278         if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
279                 return;
280         page &= 0xfffff000;
281         page += ((address>>10) & 0xffc);
    // 然后判断该页表项中的位 1（R/W）、位 0（P）标志。如果该页面不可写（R/W=0）且存在，
    // 那么就执行共享检验和复制页面操作（写时复制）。否则什么也不做，直接退出。
282         if ((3 & *(unsigned long *) page) == 1)  /* non-writeable, present */
283                 un_wp_page((unsigned long *) page);
284         return;
285 }
286
    //// 取得一页空闲内存页并映射到指定线性地址处。
    // get_free_page()仅是申请取得了主内存区的一页物理内存。而本函数则不仅是获取到一页
    // 物理内存页面，还进一步调用 put_page()，将物理页面映射到指定的线性地址处。
    // 参数 address 是指定页面的线性地址。
287 void get_empty_page(unsigned long address)
288 {
289         unsigned long tmp;
290
```

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

    // 在得到 p 进程和当前进程 address 对应的目录项后，下面分别对进程 p 和当前进程进行处理。
    // 下面首先对 p 进程的表项进行操作。目标是取得 p 进程中 address 对应的物理内存页面地址，
    // 并且该物理页面存在，而且干净（没有被修改过，不脏）。
    // 方法是先取目录项内容。如果该目录项无效（P=0），表示目录项对应的二级页表不存在，
    // 于是返回。否则取该目录项对应页表地址 from，从而计算出逻辑地址 address 对应的页表项
```

```
    // 指针，并取出该页表项内容临时保存在 phys_addr 中。
316 /* is there a page-directory at from? */
    /* 在 from 处是否存在页目录项？ */
317         from = *(unsigned long *) from_page;            // p 进程目录项内容。
318         if (!(from & 1))
319                 return 0;
320         from &= 0xfffff000;                             // 页表地址。
321         from_page = from + ((address>>10) & 0xffc);     // 页表项指针。
322         phys_addr = *(unsigned long *) from_page;       // 页表项内容。
    // 接着看看页表项映射的物理页面是否存在并且干净。 0x41 对应页表项中的 D（Dirty）和
    // P（Present）标志。如果页面不干净或无效则返回。然后我们从该表项中取出物理页面地址
    // 再保存在 phys_addr 中。最后我们再检查一下这个物理页面地址的有效性，即它不应该超过
    // 机器最大物理地址值，也不应该小于内存低端(1MB)。
323 /* is the page clean and present? */
    /* 物理页面干净并且存在吗？ */
324         if ((phys_addr & 0x41) != 0x01)
325                 return 0;
326         phys_addr &= 0xfffff000;                        // 物理页面地址。
327         if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
328                 return 0;

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

    // 在找到了进程 p 中逻辑地址 address 处对应的干净并且存在的物理页面，而且也确定了当前
    // 进程中逻辑地址 address 所对应的二级页表项地址之后，我们现在对他们进行共享处理。
    // 方法很简单，就是首先对 p 进程的页表项进行修改，设置其写保护（R/W=0，只读）标志，
    // 然后让当前进程复制 p 进程的这个页表项。此时当前进程逻辑地址 address 处页面即被
    // 映射到 p 进程逻辑地址 address 处页面映射的物理页面上。
339 /* share them: write-protect */
    /* 对它们进行共享处理：写保护 */
340         *(unsigned long *) from_page &= ~2;
341         *(unsigned long *) to_page = *(unsigned long *) from_page;
    // 随后刷新页变换高速缓冲。计算所操作物理页面的页面号，并将对应页面映射字节数组项中
    // 的引用递增 1。最后返回 1，表示共享处理成功。
342         invalidate();
343         phys_addr -= LOW_MEM;
```

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

```
363         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
364                 if (!*p)                        // 如果该任务项空闲，则继续寻找。
365                         continue;
366                 if (current == *p)              // 如果就是当前任务，也继续寻找。
367                         continue;
368                 if (address < LIBRARY_OFFSET) {
369                         if (inode != (*p)->executable)   // 进程执行文件 i 节点。
370                                 continue;
371                 } else {
372                         if (inode != (*p)->library)      // 进程使用库文件 i 节点。
373                                 continue;
374                 }
375                 if (try_to_share(address,*p))    // 尝试共享页面。
376                         return 1;
377         }
378         return 0;
379 }
380
    //// 执行缺页处理。
    // 访问不存在页面的处理函数，页异常中断处理过程中调用此函数。在 page.s 程序中被调用。
    // 函数参数 error_code 和 address 是进程在访问页面时由 CPU 因缺页产生异常而自动生成。
    // error_code 指出出错类型，参见本章开始处的“内存页面出错异常”一节；address 是产生
    // 异常的页面线性地址。
    // 该函数首先查看所缺页是否在交换设备中，若是则交换进来。否则尝试与已加载的相同文件
    // 进行页面共享，或者只是由于进程动态申请内存页面而只需映射一页物理内存页即可。若共
    // 享操作不成功，那么只能从相应文件中读入所缺的数据页面到指定线性地址处。
381 void do_no_page(unsigned long error_code,unsigned long address)
382 {
383         int nr[4];
384         unsigned long tmp;
385         unsigned long page;
386         int block,i;
387         struct m_inode * inode;
388
    // 首先判断 CPU 控制寄存器 CR2 给出的引起页面异常的线性地址在什么范围中。如果 address
    // 小于 TASK_SIZE（0x4000000，即 64MB），表示异常页面位置在内核或任务 0 和任务 1 所处
    // 的线性地址范围内，于是发出警告信息“内核范围内存被写保护”；如果 (address - 当前
    // 进程代码起始地址)大于一个进程的长度（64MB），表示 address 所指的线性地址不在引起
    // 异常的进程线性地址空间范围内，则在发出出错信息后退出。
389         if (address < TASK_SIZE)
390                 printk("\n\rBAD!! KERNEL PAGE MISSING\n\r");
391         if (address - current->start_code > TASK_SIZE) {
392                 printk("Bad things happen: nonexistent page error in do_no_page\n\r");
393                 do_exit(SIGSEGV);
394         }
    // 然后根据指定的线性地址 address 求出其对应的二级页表项指针，并根据该页表项内容判断
    // address 处的页面是否在交换设备中。若是则调入页面并退出。方法是首先取指定线性地址
    // address 对应的目录项内容。如果对应的二级页表存在，则取出该目录项中二级页表的地址，
    // 加上页表项偏移值即得到线性地址 address 处页面对应的页面表项指针，从而获得页表项内
    // 容。若页表项内容不为 0 并且页表项存在位 P=0，则说明该页表项指定的物理页面应该在交
    // 换设备中。于是从交换设备中调入指定页面后退出函数。
395         page = *(unsigned long *) ((address >> 20) & 0xffc); // 取目录项内容。
396         if (page & 1) {
```

```
397                 page &= 0xfffff000;                         // 二级页表地址。
398                 page += (address >> 10) & 0xffc;            // 页表项指针。
399                 tmp = *(unsigned long *) page;              // 页表项内容。
400                 if (tmp && !(1 & tmp)) {
401                         swap_in((unsigned long *) page);    // 从交换设备读页面。
402                         return;
403                 }
404         }
    // 否则取线性空间中指定地址 address 处页面地址，并算出指定线性地址在进程空间中相对于
    // 进程基址的偏移长度值 tmp，即对应的逻辑地址。从而可以算出缺页页面在执行文件映像中
    // 或在库文件中的具体起始数据块号。
405         address &= 0xfffff000;                              // address 处缺页页面地址。
406         tmp = address - current->start_code;                // 缺页页面对应逻辑地址。

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

```
    // page 中。
426         for (i=0 ; i<4 ; block++,i++)
427                 nr[i] = bmap(inode,block);
428         bread_page(page,inode->i_dev,nr);

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

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

    // 统计处理器分页管理逻辑页面数。页目录表前 4 项供内核代码使用，不列为统计范围，因此
    // 扫描处理的页目录项从第 5 项开始。方法是循环处理所有页目录项（除前 4 个项），若对应
    // 的二级页表存在，那么先统计二级页表本身占用的内存页面（484 行），然后对该页表中所
    // 有页表项对应页面情况进行统计。
475         k = 0;                                  // 一个进程占用页面统计值。
476         for(i=4 ; i<1024 ;) {
477                 if (1&pg_dir[i]) {
    // （如果页目录项对应二级页表地址大于机器最高物理内存地址 HIGH_MEMORY，则说明该目录项
    // 有问题。于是显示该目录项信息并继续处理下一个目录项。）
478                         if (pg_dir[i]>HIGH_MEMORY) {    // 目录项内容不正常。
479                                 printk("page directory[%d]: %08X\n\r",
480                                         i,pg_dir[i]);
481                                 continue;               // continue 之前需插入 i++;
482                         }
    // 如果页目录项对应二级页表的“地址”大于 LOW_MEM（即 1MB），则把一个进程占用的物理
    // 内存页统计值 k 增 1，把系统占用的所有物理内存页统计值 free 增 1。然后取对应页表地址
    // pg_tbl，并对该页表中所有页表项进行统计。如果当前页表项所指物理页面存在并且该物理
    // 页面“地址”大于 LOW_MEM，那么就将页表项对应页面纳入统计值。
483                         if (pg_dir[i]>LOW_MEM)
484                                 free++,k++;             // 统计页表占用页面。
```

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