




第 10 章 内存管理(mm)

10.1 概述

内存管理目录中共有三个程序，如下表所示：

列表 10.1 内存管理子目录文件列表

	Name	Size	Last modified (GMT)	Description
	Makefile	813 bytes	1991-12-02 03:21:45	m
	memory.c	11223 bytes	1991-12-03 00:48:01	m
	page.s	508 bytes	1991-10-02 14:16:30	m

下面对每一个文件分别描述。

10.2 Makefile 文件

10.2.1 功能描述

10.2.2 代码注释

列表 10.2 linux/mm/Makefile 文件

```
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。
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 OBJS    = memory.o page.o    # 定义目标文件变量 OBJS。
19
20 all: mm.o
21
22 mm.o: $(OBJS)          # 在有了先决条件 OBJS 后使用下面的命令连接成目标 mm.o
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

```

10.3 memory.c 程序

10.3.1 功能描述

10.3.2 代码注释

列表 linux/mm/memory.c 程序

```

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 - 在程序编制表中是呼是最重要的程序，
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。
29   *
30   * 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
31   * 6M 的内存，而目前却不用。目前看来工作的很好。
32   *
33   * 对"invalidate()"函数也进行了修正 - 在这方面我还做的不够。
34   */
35
36  #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。
37
38  #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。
39
40  #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，
41                               // 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。
42
43  #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。
44
45  #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
46
47  volatile void do_exit(long code);
48
49  //// 内存已用完。
50
51  static inline volatile void oom(void)
52  {
53      printk("out of memory\n\r");

```

```

36     do_exit(SIGSEGV);
37 }
38
// 刷新页变换高速缓冲。
// 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。
39 #define invalidate() \
40 __asm__ ("movl %%eax, %%cr3": "a" (0))
41
42 /* these are not to be changed without changing head.s etc */
// 下面代码需要与 head.s 等中的相关信息一起改变 */
43 #define LOW_MEM 0x100000 // 内存低端（1MB）。
44 #define PAGING_MEMORY (15*1024*1024) // 分页内存 15MB。
45 #define PAGING_PAGES (PAGING_MEMORY>>12) // 分页后的页数。
46 #define MAP_NR(addr) (((addr)-LOW_MEM)>>12) // 映射页号。
47 #define USED 100
48
// CODE_SPACE(addr) (((addr)+0xfff)&~0xfff) < current->start_code + current->end_code。
49 #define CODE_SPACE(addr) (((addr)+4095)&~4095) < \
50 current->start_code + current->end_code)
51
52 static long HIGH_MEMORY = 0;
53
// 复制页面（4K 字节）。
54 #define copy_page(from,to) \
55 __asm__ ("cld ; rep ; movsl": "S" (from), "D" (to), "c" (1024): "cx", "di", "si")
56
57 static unsigned char mem_map [ PAGING_PAGES ] = {0,}; // 内存映射字节图(1 字节代表 1 页)。
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。如果没有空闲页面了，
// 就返回 0。
// 取空闲页面。如果已经没有内存了，则返回 0。
// 输入：%1(ax=0) - 0；%2(LOW_MEM)；%3(cx=PAGING_PAGES)；%4(di=mem_map+PAGING_PAGES-1)。
// 输出：返回%0(ax=页面号)。
// 从内存映像末端开始向前扫描所有页面标志（页面总数为 PAGING_PAGES），如果有页面空闲（对应
// 内存映像位为 0）则返回页面地址。
63 unsigned long get_free_page(void)
64 {
65     register unsigned long __res asm("ax");
66
67     __asm__ ("std ; repne ; scasb\n\t" // 方向位置位，将 a1(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" // 将对应页面的内存映像位置 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，即获得页面实际起始地址。
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, 也即将该页面清零)。
76     "movl %%edx, %%eax\n\t"     // 将页面起始地址→eax (返回值)。
77     "l:"
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;                  // 返回空闲页面地址 (如果无空闲也则返回 0)。
83 }
84
85 /*
86  * Free a page of memory at physical address 'addr'. Used by
87  * 'free_page_tables()'
88  */
89
90 // 释放物理地址 'addr' 的页面。用于 'free_page_tables()'。
91 // 1MB 以下的内存空间用于操作系统 (内核), 不作为分配页面的内存。
92 void free_page(unsigned long addr)
93 {
94     if (addr < LOW_MEM) return; // 如果物理地址 addr 小于内存低端 (1MB), 则返回。
95     if (addr >= HIGH_MEMORY)    // 如果物理地址 addr ≥ 内存高端, 则显示出错信息。
96         panic("trying to free nonexistent page");
97     addr -= LOW_MEM;             // 物理地址-低端内存位置, 再除以 4KB, 得页面号。
98     addr >>= 12;
99     if (mem_map[addr]--) return; // 如果对应内存页面映射字节不等于 0, 则减 1 返回。
100    mem_map[addr]=0;             // 否则置对应页面映射字节为 0, 并显示出错信息, 死机。
101    panic("trying to free free page");
102 }
103
104 /*
105  * This function frees a continuous block of page tables, as needed
106  * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
107  */
108
109 // 下面函数释放页表连续的内存块, 'exit()' 需要该函数。与 copy_page_tables()
110 // 类似, 该函数仅处理 4Mb 的内存块。
111
112 // 根据指定的地址和限长, 释放对应内存页表所指定的内存块及页表本身。
113 // 页目录位于物理地址 0 开始处, 共 1024 项, 占 4K 字节。每个目录项指定一个页表。
114 // 页表从物理地址 0x1000 处开始 (紧接着目录空间), 每个页表有 1024 项, 也占 4K 内存。
115 // 每个页表项指定一页内存 (4K)。目录项和页表项的大小均为 4 个字节。
116 // 参数: from - 起始基地址; size - 释放的长度。
117 int free_page_tables(unsigned long from, unsigned long size)
118 {
119     unsigned long *pg_table;
120     unsigned long *dir, nr;
121
122     if (from & 0x3fffff) // 要释放内存块的地址需以 4M 为边界。
123         panic("free_page_tables called with wrong alignment");
124     if (!from)           // 出错, 试图释放内核所占空间。

```

```

113         panic("Trying to free up swapper memory space");
// 计算所占页目录项数(4M 的进位整数倍), 也即所占页表数。
114     size = (size + 0x3ffff) >> 22;
// 下面一句计算起始目录项。对应的目录项号=from>>22, 因每项占 4 字节, 又由于页目录是从
// 物理地址 0 开始, 因此实际的目录项指针=目录项号<<2, 也即(from>>20)。与上 0xffc 确保
// 目录项指针范围有效。
115     dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
116     for ( ; size-->0 ; dir++) { // size 现在是需释放内存的目录项数。
117         if (!(1 & *dir)) // 如果该目录项无效(P 位=0), 则继续。
118             continue;
119         pg_table = (unsigned long *) (0xfffff000 & *dir); // 取目录项中页表地址。
120         for (nr=0 ; nr<1024 ; nr++) { // 每个页表有 1024 个页项。
121             if (1 & *pg_table) // 如果该页表项有效(P 位=1), 则释放该页。
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; // 对相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 /*
151  * 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
152  * 来拷贝一定范围内的线性地址中的内容。希望代码中没有错误, 因为我不想
153  * 再调试这块代码了☺。
154  *
155  * 注意! 我们并不是仅复制任何内存块 - 内存块的地址需要是 4Mb 的倍数 (正好
156  * 一个页目录项对应的内存大小), 因为这样处理可使函数很简单。不管怎样,
157  * 它仅被 fork() 使用。
158  *
159  * 注意 2!! 当 from==0 时, 是在为第一次 fork() 调用复制内核空间。此时我们
160  * 不想复制整个页目录项对应的内存, 因为这样做会导致严重的内存浪费 - 我们
161  * 只复制头 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。

*/

//// 复制进程的页目录页表。

```

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
158     // 源地址和目的地址都需要是 4Mb 的倍数。否则出错, 死机。
159     if ((from&0x3fffff) || (to&0x3fffff))
160         panic("copy_page_tables called with wrong alignment");
161     // 取得源地址和目的地址的目录项(from_dir 和 to_dir)。
162     from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
163     to_dir = (unsigned long *) ((to>>20) & 0xffc);
164     // 计算要复制的内存块占用的页表数 (也即目录项数)。
165     size = ((unsigned) (size+0x3fffff)) >> 22;
166     // 下面开始对每个占用的页表依次进行复制操作。
167     for( ; size-->0 ; from_dir++,to_dir++) {
168         // 如果目的目录项指定的页表已经存在(P=1), 则出错, 死机。
169         if (1 & *to_dir)
170             panic("copy_page_tables: already exist");
171         // 如果此源目录项未被使用, 则不用复制对应页表, 跳过。
172         if (!(1 & *from_dir))
173             continue;
174         // 取当前源目录项中页表的地址 → from_page_table。
175         from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
176         // 为目的页表取一页空闲内存, 如果返回是 0 则说明没有申请到空闲内存页面。返回值=-1, 退出。
177         if (!(to_page_table = (unsigned long *) get_free_page()))
178             return -1; /* Out of memory, see freeing */
179         // 设置目的目录项信息。7 是标志信息, 表示(Usr, R/W, Present)。
180         *to_dir = ((unsigned long) to_page_table) | 7;
181         // 针对当前处理的页表, 设置需复制的页面数。如果是在内核空间, 则仅需复制头 160 页, 否则需要
182         // 复制 1 个页表中的所有 1024 页面。
183         nr = (from==0)?0xA0:1024;
184         // 对于当前页表, 开始复制指定数目 nr 个内存页面。
185         for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
186             this_page = *from_page_table; // 取源页表项内容。
187             if (!(1 & this_page)) // 如果当前源页面没有使用, 则不用复制。
188                 continue;
189             // 复位页表项中 R/W 标志(置 0)。(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,
190             // 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, 则就有写的权限。)
191             this_page &= ~2;
192             *to_page_table = this_page; // 将该页表项复制到目的页表中。
193             // 如果该页表项所指页面的地址在 1M 以上, 则需要设置内存页面映射数组 mem_map[], 于是计算
194             // 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。
195             if (this_page > LOW_MEM) {
196                 *from_page_table = this_page; // 令源页表项也只读[??]。
197                 this_page -= LOW_MEM;
198                 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  */
197 /*
198  * 下面函数将一页面放置在内存中指定地址处。它返回页面的物理地址，如果
199  * 内存不够(在访问页表或页面时)，则返回 0。
200  */
201 // 在指定物理地址处放置一页面。
202 // 主要工作是在页目录和页表中设置指定页面的信息。
203 // 若成功则返回页面地址。
204 unsigned long put_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     /* 注意!!!这里使用了页目录基址_pg_dir=0 的条件 */
210
211     // 如果申请的页面位置低于 LOW_MEM(1Mb)或超出系统实际含有内存高端 HIGH_MEMORY，则发出警告。
212     if (page < LOW_MEM || page >= HIGH_MEMORY)
213         printk("Trying to put page %p at %p\n", page, address);
214     // 如果申请的页面在内存页面映射字节图中没有置位，则显示警告信息。
215     if (mem_map[(page-LOW_MEM)>>12] != 1)
216         printk("mem_map disagrees with %p at %p\n", page, address);
217     // 计算指定地址在页目录中对应的目录项指针。
218     page_table = (unsigned long *) ((address>>20) & 0xffc);
219     // 如果该目录项有效(P=1)(也即指定的页表在内存中)，则从中取得指定页表的地址→page_table。
220     if ((*page_table)&1)
221         page_table = (unsigned long *) (0xfffff000 & *page_table);
222     else {
223         // 否则，申请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
224         // 该页表的地址→page_table。
225         if (!(tmp=get_free_page()))
226             return 0;
227         *page_table = tmp|7;
228         page_table = (unsigned long *) tmp;
229     }
230     // 在页表中设置指定地址页面的页表项内容。每个页表共可有 1024 项(0x3ff)。
231     page_table[(address>>12) & 0x3ff] = page | 7;
232     /* no need for invalidate */
233     /* 不需要刷新页变换高速缓冲 */
234     return page;           // 返回页面地址。
235 }

```



```

220 220
221 221 // 取消写保护页面函数（写时复制）。
222 222 // 输入参数为页表项指针。
223 223 // [?? un_wp_page 意思是否是取消页面的写保护?? wp -- Write Protected。]
224 224 void un_wp_page(unsigned long * table_entry)
225 225 {
226 226     unsigned long old_page, new_page;
227 227
228 228     old_page = 0xfffff000 & *table_entry; // 取原页面地址。
229 229 // 如果原页面地址大于内存低端 LOW_MEM(1Mb) 并且其在页面映射字节数组中值为 1（表示仅被
230 230 // 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
231 231 // 高速缓冲，然后返回。
232 232 if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
233 233     *table_entry |= 2;
234 234     invalidate();
235 235     return;
236 236 }
237 237 // 否则，申请一页空闲页面。
238 238 if (!(new_page=get_free_page()))
239 239     oom(); // Out of Memory。内存不够处理。
240 240 // 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
241 241 // 数组值递减 1。然后将指定页表项内容更新为新页面的地址，并置可读写等标志 (U/S, R/W, P)。
242 242 // 刷新页变换高速缓冲。最后将原页面内容复制到新页面。
243 243 if (old_page >= LOW_MEM)
244 244     mem_map[MAP_NR(old_page)]--;
245 245 *table_entry = new_page | 7;
246 246 invalidate();
247 247 copy_page(old_page, new_page);
248 248 }
249 249
250 250 /*
251 251  * This routine handles present pages, when users try to write
252 252  * to a shared page. It is done by copying the page to a new address
253 253  * and decrementing the shared-page counter for the old page.
254 254  *
255 255  * If it's in code space we exit with a segment error.
256 256  */
257 257 /*
258 258  * 当用户试图往一个共享页面上写时，该函数处理已存在的内存页面，（写时复制）
259 259  * 它是通过将页面复制到一个新地址上并递减原页面的共享页面计数值实现的。
260 260  *
261 261  * 如果它在代码空间，我们就以段错误信息退出。
262 262  */
263 263 // 页面异常中断处理调用函数。写共享页面处理函数。
264 264 // 参数 error_code 是由 CPU 自动产生，address 是页面线性地址。
265 264 // 写共享页面时，需复制页面（写时复制）。
266 266 void do_wp_page(unsigned long error_code, unsigned long address)
267 267 {
268 268 #if 0
269 269 /* we cannot do this yet: the estdio library writes to code space */
270 270 /* stupid, stupid. I really want the libc.a from GNU */
271 271 /* 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 */
272 271 /* 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
// 处理取消页面保护。参数指定页面在页表中的页表项指针，其计算方法是：
// ((address>>10) & 0xffc)：计算指定地址的页面在页表中的偏移地址；
// (0xfffff000 & ((address>>20) & 0xffc))：取目录项中页表的地址值，
// 其中((address>>20) & 0xffc)计算页面所在页表的目录项指针；
// 两者相加即得指定地址对应页面的页表项指针。
255         un_wp_page((unsigned long *)
256                     (((address>>10) & 0xffc) + (0xfffff000 &
257                                                     *((unsigned long *) ((address>>20) & 0xffc)))));
258
259     }
260
261     // 写页面验证。
262 void write_verify(unsigned long address)
263 {
264     unsigned long page;
265
266     // 判断指定地址所对应的页表页面是否存在(P)，若不存在(P=0)则返回。
267     if (!(page = *((unsigned long *) ((address>>20) & 0xffc)) & 1))
268         return;
269     // 取页表的地址，加上指定地址的页面在页表中的页表项偏移值，得对应页面的页表项指针。
270     page &= 0xfffff000;
271     page += ((address>>10) & 0xffc);
272     // 如果该页面不可写(标志 R/W 没有置位)，则执行共享检验和复制页面操作(写时复制)。
273     if ((3 & *((unsigned long *) page) == 1) /* non-writeable, present */
274         un_wp_page((unsigned long *) page);
275     return;
276 }
277
278 // 取指定地址的空页面。
279 // 与 get_free_page() 不同。get_free_page() 仅置内存页面映射数组 mem_map[] 中的引用标志。
280 // 而这里 get_empty_page() 不仅是取指定地址处的页面，还进一步调用 put_page()，将页面信息
281 // 添加到页目录和页表中。
282 void get_empty_page(unsigned long address)
283 {
284     unsigned long tmp;
285
286     if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
287         free_page(tmp);          /* 0 is ok - ignored */
288         oom();
289     }
290 }
291
292 /*
293  * try_to_share() checks the page at address "address" in the task "p",
294  * to see if it exists, and if it is clean. If so, share it with the current
295  * task.
296  *
297  * NOTE! This assumes we have checked that p != current, and that they
298  * share the same executable.
299  */

```

```

/*
 * 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。
 */
///// 尝试共享。
// 返回 1-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
    // 求指定内存地址的页目录项。
300     from_page = to_page = ((address>>20) & 0xffc);
    // 计算地址在指定进程 p 中对应的页目录项。
301     from_page += ((p->start_code>>20) & 0xffc);
    // 计算地址在当前进程中对应的页目录项。
302     to_page += ((current->start_code>>20) & 0xffc);
303 /* is there a page-directory at from? */
    // 在 from 处是否存在页目录？*/
    // *** 对 p 进程页面进行操作。
    // 取页目录项内容。如果该目录项无效(P=0)，则返回。否则取该目录项对应页表地址→from。
304     from = *(unsigned long *) from_page;
305     if (!(from & 1))
306         return 0;
307     from &= 0xfffff000;
    // 计算地址对应的页表项指针值，并取出该页表项内容→phys_addr。
308     from_page = from + ((address>>10) & 0xffc);
309     phys_addr = *(unsigned long *) from_page;
310 /* is the page clean and present? */
    // 页面干净并且存在吗？*/
    // 0x41 对应页表项中的 Dirty 和 Present 标志。如果页面不干净或无效则返回。
311     if ((phys_addr & 0x41) != 0x01)
312         return 0;
    // 取页面的地址→phys_addr。如果该页面地址不存在或小于内存低端(1M)也返回退出。
313     phys_addr &= 0xfffff000;
314     if (phys_addr >= HIGH MEMORY || phys_addr < LOW MEM)
315         return 0;
    // *** 对当前进程页面进行操作。
    // 取页目录项内容→to。如果该目录项无效(P=0)，则取空闲页面，并让更新 to_page 所指的目录项。
316     to = *(unsigned long *) to_page;
317     if (!(to & 1))
318         if (to = get\_free\_page())
319             *(unsigned long *) to_page = to | 7;
320     else
321         oom();
    // 取对应页表地址→to，页表项地址→to_page。如果对应的页面已经存在，则出错，死机。
322     to &= 0xfffff000;
323     to_page = to + ((address>>10) & 0xffc);

```

```

324         if (1 & *(unsigned long *) to_page)
325             panic("try_to_share: to_page already exists");
326 /* share them: write-protect */
327 /* 对它们进行共享处理：写保护 */
328 // 对 p 进程中页面置写保护标志(置 R/W=0)。并且当前进程中的对应页表项指向它。
329     *(unsigned long *) from_page &= ~2;
330     *(unsigned long *) to_page = *(unsigned long *) from_page;
331 // 刷新页变换高速缓冲。
332     invalidate();
333 // 计算所操作页面的页面号，并将对应页面映射数组项中的引用递增 1。
334     phys_addr -= LOW\_MEM;
335     phys_addr >>= 12;
336     mem\_map[phys_addr]++;
337     return 1;
338 }
339
340 /*
341 * share_page() tries to find a process that could share a page with
342 * the current one. Address is the address of the wanted page relative
343 * to the current data space.
344 *
345 * We first check if it is at all feasible by checking executable->i_count.
346 * It should be >1 if there are other tasks sharing this inode.
347 */
348 /*
349 * 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
350 * 当前数据空间中期望共享的某页面地址。
351 *
352 * 首先我们通过检测 executable->i_count 来查证是否可行。如果有其它任务已共享
353 * 该 inode，则它应该大于 1。
354 */
355 //// 共享页面。
356 // 返回 1 - 成功，0 - 失败。
357 static int share\_page(unsigned long address)
358 {
359     struct task\_struct ** p;
360
361     // 如果是不可执行的，则返回。
362     if (!current->executable)
363         return 0;
364     // 如果只能单独执行(executable->i_count=1)，也退出。
365     if (current->executable->i_count < 2)
366         return 0;
367     // 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，并尝试对指定地址的页面进行共享。
368     for (p = &LAST\_TASK ; p > &FIRST\_TASK ; --p) {
369         if (!*p) // 如果该任务项空闲，则继续寻找。
370             continue;
371         if (current == *p) // 如果就是当前任务，也继续寻找。
372             continue;
373         if ((*p)->executable != current->executable) // 如果 executable 不等，也继续。
374             continue;
375         if (try\_to\_share(address, *p)) // 尝试共享页面。
376             return 1;

```

```

361     }
362     return 0;
363 }
364
365 // 页面异常中断处理调用函数。缺页异常处理函数。
366 // 参数 error_code 是由 CPU 自动产生，address 是页面线性地址。
367 void do_no_page(unsigned long error_code, unsigned long address)
368 {
369     int nr[4];
370     unsigned long tmp;
371     unsigned long page;
372     int block, i;
373
374     address &= 0xfffff000; // 页面地址。
375     // 如果当前进程的 executable 空，或者指定地址超出进程的数据末端，则取指定地址空页面并退出。
376     tmp = address - current->start_code;
377     if (!current->executable || tmp >= current->end_data) {
378         get_empty_page(address);
379         return;
380     }
381     // 如果在所有进程中尝试共享页面失败，则退出。
382     if (share_page(tmp))
383         return;
384     // 取空闲页面，如果内存不够了，则进行内存不够处理。
385     if (!(page = get_free_page()))
386         oom();
387     /* remember that 1 block is used for header */
388     /* 记住，（任务）头要使用 1 个数据块 */
389     //
390     block = 1 + tmp/BLOCK_SIZE;
391     for (i=0 ; i<4 ; block++, i++)
392         nr[i] = bmap(current->executable, block);
393     bread_page(page, current->executable->i_dev, nr);
394     i = tmp + 4096 - current->end_data;
395     tmp = page + 4096;
396     while (i-- > 0) {
397         tmp--;
398         *(char *)tmp = 0;
399     }
400     if (put_page(page, address))
401         return;
402     free_page(page);
403     oom();
404 }
405
406 // 内存初始化。
407 // 参数：start_mem - 可用作分页处理的物理内存起始位置（已去除 RAMDISK 所占内存空间等）。
408 //       end_mem   - 实际物理内存最大地址。
409 // 在该版的 linux 内核中，最多能使用 16Mb 的内存，大于 16Mb 的内存将不予考虑，弃置不用。
410 // 0 - 1Mb 内存空间用于内核系统（其实是 0-640Kb）。
411 void mem_init(long start_mem, long end_mem)
412 {
413     int i;

```

```

402
403     HIGH_MEMORY = end_mem;           // 设置内存最高端。
404     for (i=0 ; i<PAGING_PAGES ; i++) // 首先置所有页面为已占用 (USED=100) 状态,
405         mem_map[i] = USED;           // 即将页面映射数组全置成 USED。
406     i = MAP_NR(start_mem);           // 然后计算可使用起始内存的页面号。
407     end_mem -= start_mem;             // 再计算可分页处理的内存块大小。
408     end_mem >>= 12;                   // 从而计算出可用于分页处理的页面数。
409     while (end_mem-->0)               // 最后将这些可用页面对应的页面映射数组清零。
410         mem_map[i++]=0;
411 }
412
413 // 计算内存空闲页面数并显示。
414 void calc_mem(void)
415 {
416     int i, j, k, free=0;
417     long * pg_tbl;
418
419     // 扫描内存页面映射数组 mem_map[], 获取空闲页面数并显示。
420     for(i=0 ; i<PAGING_PAGES ; i++)
421         if (!mem_map[i]) free++;
422     printk("%d pages free (of %d)\n", free, PAGING_PAGES);
423 // 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。
424     for(i=2 ; i<1024 ; i++) {
425         if (l&pg_dir[i]) {
426             pg_tbl=(long *) (0xffffffff & pg_dir[i]);
427             for(j=k=0 ; j<1024 ; j++)
428                 if (pg_tbl[j]&1)
429                     k++;
430             printk("Pg-dir[%d] uses %d pages\n", i, k);
431         }
432     }

```

10.4 page.s 程序

10.4.1 功能描述

10.4.2 代码注释

列表 linux/mm/page.s 程序

```

1 /*
2  * linux/mm/page.s
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*

```

```

8  * page.s contains the low-level page-exception code.
9  * the real work is done in mm.c
10 */
/*
   * page.s 程序包含底层页异常处理代码。实际的工作在 memory.c 中完成。
   */
11
// 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；二是其它页保护引起的页异常，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。
12 .globl _page_fault
13
14 _page_fault:
15     xchgl %eax, (%esp)    # 取出错码到 eax。
16     pushl %ecx
17     pushl %edx
18     push %ds
19     push %es
20     push %fs
21     movl $0x10, %edx     # 置内核数据段选择符。
22     mov %dx, %ds
23     mov %dx, %es
24     mov %dx, %fs
25     movl %cr2, %edx      # 取引起页面异常的线性地址
26     pushl %edx           # 将该线性地址和出错码压入堆栈，作为调用函数的参数。
27     pushl %eax
28     testl $1, %eax      # 测试标志 P，如果不是缺页引起的异常则跳转。
29     jne 1f
30     call _do_no_page     # 调用缺页处理函数（mm/memory.c, 365 行）。
31     jmp 2f
32 1:    call _do_wp_page    # 调用写保护处理函数（mm/memory.c, 247 行）。
33 2:    addl $8, %esp       # 丢弃压入栈的两个参数。
34     pop %fs
35     pop %es
36     pop %ds
37     popl %edx
38     popl %ecx
39     popl %eax
40     iret

```

10.4.3 其它信息

10.4.3.1 页异常的处理

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。