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

/*
 * This file should contain most things doing the swapping from/to disk.
 * Started 18.12.91
 */
/*
 * 本程序应该包括绝大部分执行内存交换的代码（从内存到磁盘或反之）。
 * 从 91 年 12 月 18 日开始编制。
 */

#include <string.h>        // 字符串头文件。定义了一些有关内存或字符串操作的嵌入函数。

#include <linux/mm.h>      // 内存管理头文件。定义页面长度，和一些内存管理函数原型。
#include <linux/sched.h>   // 调度程序头文件。定义了任务结构 task_struct、任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/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。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原型定义。

// 每个字节 8 位，因此 1 页（4096 字节）共有 32768 个比特位。若 1 个比特位对应 1 页内存，
// 则最多可管理 32768 个页面，对应 128MB 内存容量。
#define SWAP_BITS (4096<<3)

// 比特位操作宏。通过给定不同的"op"，可定义对指定比特位进行测试、设置或清除三种操作。
// 参数 addr 是指定线性地址；nr 是指定地址处开始的比特位偏移位。该宏把给定地址 addr 处
// 第 nr 个比特位的值放入进位标志，设置或复位该比特位并返回进位标志值（即原比特位值）。
// 第 25 行上第一个指令随"op"字符的不同而组合形成不同的指令：
// 当"op"= ""时，就是指令 bt  -  （Bit Test）测试并用原值设置进位位。
// 当"op"="s"时，就是指令 bts -  （Bit Test and Set）设置比特位值并用原值设置进位位。
// 当"op"="r"时，就是指令 btr -  （Bit Test and Reset）复位比特位值并原值设置进位位。
// 输入：%0  -  （返回值），%1 -位偏移(nr)；%2  - 基址(addr)；%3  -  加操作寄存器初值(0)。
// 内嵌汇编代码把基地址（%2）和比特偏移值（%1）所指定的比特位值先保存到进位标志 CF 中，
// 然后设置（复位）该比特位。指令 adcl 是带进位位加，用于根据进位位 CF 设置操作数（%0）。
// 如果 CF = 1 则返回寄存器值 = 1，否则返回寄存器值 = 0 。
#define bitop(name,op) \
static inline int name(char * addr,unsigned int nr) \
{ \
int __res; \
__asm__ __volatile__("bt" op " %1,%2; adcl $0,%0" \
:"=g" (__res) \
:"r" (nr),"m" (*(addr)),"0" (0)); \
return __res; \
}

// 这里根据不同的 op 字符定义 3 个内嵌函数。
bitop(bit,"")                // 定义内嵌函数 bit(char * addr, unsigned int nr)。
bitop(setbit,"s")            // 定义内嵌函数 setbit(char * addr, unsigned int nr)。
bitop(clrbit,"r")            // 定义内嵌函数 clrbit(char * addr, unsigned int nr)。
```

```
34
35 static char * swap_bitmap = NULL;
36 int SWAP_DEV = 0;                              // 内核初始化时设置的交换设备号。
37
38 /*
39  * We never page the pages in task[0] - kernel memory.
40  * We page all other pages.
41  */
   /*
    * 我们从不交换任务 0（task[0]）的页面 - 即不交换内核页面。
    * 我们只对其他页面进行交换操作。
    */
   // 第 1 个虚拟内存页面。即从任务 0 末端（64MB）处开始的虚拟内存页面。
42 #define FIRST_VM_PAGE (TASK_SIZE>>12)              // = 64MB/4KB = 16384。
43 #define LAST_VM_PAGE (1024*1024)                   // = 4GB/4KB  = 1048576。
44 #define VM_PAGES (LAST_VM_PAGE - FIRST_VM_PAGE)    // = 1032192（从 0 开始计）。
45
   // 申请 1 页交换页面。
   // 扫描整个交换映射位图（除对应位图本身的位 0 以外），返回值为 1 的第一个比特位号，
   // 即目前空闲的交换页面号。若操作成功则返回交换页面号，否则返回 0。
46 static int get_swap_page(void)
47 {
48         int nr;
49
50         if (!swap_bitmap)
51                 return 0;
52         for (nr = 1; nr < 32768 ; nr++)
53                 if (clrbit(swap_bitmap,nr))
54                         return nr;                 // 返回目前空闲的交换页面号。
55         return 0;
56 }
57
   // 释放交换设备中指定的交换页面。
   // 在交换位图中设置指定页面号对应的比特位（置 1）。若原来该比特位就等于 1，则表示
   // 交换设备中原来该页面就没有被占用，或者位图出错。于是显示出错信息并返回。
   // 参数指定交换页面号。
58 void swap_free(int swap_nr)
59 {
60         if (!swap_nr)
61                 return;
62         if (swap_bitmap && swap_nr < SWAP_BITS)
63                 if (!setbit(swap_bitmap,swap_nr))
64                         return;
65         printk("Swap-space bad (swap_free())\n\r");
66         return;
67 }
68
   // 把指定页面交换进内存中。
   // 把指定页表项的对应页面从交换设备中读入到新申请的内存页面中。修改交换位图中对应
   // 比特位（置位），同时修改页表项内容，让它指向该内存页面，并设置相应标志。
69 void swap_in(unsigned long *table_ptr)
70 {
71         int swap_nr;
```

```
72          unsigned long page;
73
    // 首先检查交换位图和参数有效性。如果交换位图不存在，或者指定页表项对应的页面已存在
    // 于内存中，或者交换页面号为 0，则显示警告信息并退出。对于已放到交换设备中去的内存
    // 页面，相应页表项中存放的应是交换页面号*2，即(swap_nr << 1)，参见下面对尝试交换函
    // 数 try_to_swap_out()中第 111 行的说明。
74          if (!swap_bitmap) {
75                  printk("Trying to swap in without swap bit-map");
76                  return;
77          }
78          if (1 & *table_ptr) {
79                  printk("trying to swap in present page\n\r");
80                  return;
81          }
82          swap_nr = *table_ptr >> 1;
83          if (!swap_nr) {
84                  printk("No swap page in swap_in\n\r");
85                  return;
86          }
    // 然后申请一页物理内存并从交换设备中读入页面号为 swap_nr 的页面。在把页面交换进来
    // 后，就把交换位图中对应比特位置位。如果其原本就是置位的，说明此次是再次从交换设
    // 备中读入相同的页面，于是显示一下警告信息。最后让页表项指向该物理页面，并设置页
    // 面已修改、用户可读写和存在标志（Dirty、U/S、R/W、P）。
87          if (!(page = get_free_page()))
88                  oom();
89          read_swap_page(swap_nr, (char *) page);  // 在 include/linux/mm.h 中定义。
90          if (setbit(swap_bitmap,swap_nr))
91                  printk("swapping in multiply from same page\n\r");
92          *table_ptr = page | (PAGE_DIRTY | 7);
93  }
94
    // 尝试把页面交换出去。
    // 若页面没有被修改过则不用保存在交换设备中，因为对应页面还可以再直接从相应映像文件
    // 中读入。于是可以直接释放掉相应物理页面了事。否则就申请一个交换页面号，然后把页面
    // 交换出去。此时交换页面号要保存在对应页表项中，并且仍需要保持页表项存在位 P = 0。
    // 参数是页表项指针。页面交换或释放成功返回 1，否则返回 0。
95  int try_to_swap_out(unsigned long * table_ptr)
96  {
97          unsigned long page;
98          unsigned long swap_nr;
99
    // 首先判断参数的有效性。若需要交换出去的内存页面并不存在（或称无效），则即可退出。
    // 若页表项指定的物理页面地址大于分页管理的内存高端 PAGING_MEMORY（15MB），也退出。
100         page = *table_ptr;
101         if (!(PAGE_PRESENT & page))
102                 return 0;
103         if (page - LOW_MEM > PAGING_MEMORY)
104                 return 0;
    // 若内存页面已被修改过，但是该页面是被共享的，那么为了提高运行效率，此类页面不宜
    // 被交换出去，于是直接退出，函数返回 0。否则就申请一交换页面号，并把它保存在页表
    // 项中，然后把页面交换出去并释放对应物理内存页面。
105         if (PAGE_DIRTY & page) {
106                 page &= 0xfffff000;                     // 取物理页面地址。
```

```
107                 if (mem_map[MAP_NR(page)] != 1)
108                         return 0;
109                 if (!(swap_nr = get_swap_page()))       // 申请交换页面号。
110                         return 0;
    // 对于要到交换设备中的页面，相应页表项中将存放的是(swap_nr << 1)。 乘 2（左移 1 位）
    // 是为了空出原来页表项的存在位（P）。只有存在位 P=0 并且页表项内容不为 0 的页面才会
    // 在交换设备中。 Intel 手册中明确指出，当一个表项的存在位 P = 0 时（无效页表项），
    // 所有其他位（位 31—1）可供随意使用。下面写交换页函数 write_swap_page(nr, buffer)
    // 被定义为 ll_rw_page(WRITE,SWAP_DEV,(nr),(buffer))。参见 linux/mm.h 文件第 12 行。
111                 *table_ptr = swap_nr<<1;
112                 invalidate();                           // 刷新 CPU 页变换高速缓冲。
113                 write_swap_page(swap_nr, (char *) page);
114                 free_page(page);
115                 return 1;
116         }
    // 否则表明页面没有修改过。那么就不用交换出去，而直接释放即可。
117         *table_ptr = 0;
118         invalidate();
119         free_page(page);
120         return 1;
121 }
122
123 /*
124  * Ok, this has a rather intricate logic - the idea is to make good
125  * and fast machine code. If we didn't worry about that, things would
126  * be easier.
127  */
    /*
     * OK，这个函数中有一个非常复杂的逻辑 - 用于产生逻辑性好并且速度快的
     * 机器码。如果我们不对此操心的话，那么事情可能更容易些。
     */
    // 把内存页面放到交换设备中。
    // 从线性地址 64MB 对应的目录项（FIRST_VM_PAGE>>10）开始，搜索整个 4GB 线性空间，对有
    // 效页目录二级页表的页表项指定的物理内存页面执行交换到交换设备中去的尝试。一旦成功
    // 地换出一个页面，就返回 1。否则返回 0。该函数会在 get_free_page()中被调用。
128 int swap_out(void)
129 {
130         static int dir_entry = FIRST_VM_PAGE>>10;   // 即任务 1 的第 1 个目录项索引。
131         static int page_entry = -1;
132         int counter = VM_PAGES;
133         int pg_table;
134
    // 首先搜索页目录表，查找二级页表存在的页目录项 pg_table。找到则退出循环，否则调整
    // 页目录项数对应剩余二级页表项数 counter，然后继续检测下一页目录项。 若全部搜索完
    // 还没有找到适合的（存在的）页目录项，就重新继续搜索。
135         while (counter>0) {
136                 pg_table = pg_dir[dir_entry];           // 页目录项内容。
137                 if (pg_table & 1)
138                         break;
139                 counter -= 1024;                        // 1 个页表对应 1024 个页帧。
140                 dir_entry++;                            // 下一目录项。
141                 if (dir_entry >= 1024)
142                         dir_entry = FIRST_VM_PAGE>>10;
```

```
143         }
    // 在取得当前目录项的页表指针后，针对该页表中的所有 1024 个页面，逐一调用交换函数
    // try_to_swap_out()尝试交换出去。一旦某个页面成功交换到交换设备中就返回 1。若对所
    // 有目录项的所有页表都已尝试失败，则显示“交换内存用完”的警告，并返回 0。
144         pg_table &= 0xfffff000;                         // 页表指针（地址）。
145         while (counter-- > 0) {
146                 page_entry++;                           // 页表项索引（初始为-1）。
    // 如果已经尝试处理完当前页表所有项还没有能够成功地交换出一个页面，即此时页表项索引
    // 大于等于 1024，则如同前面第 135 - 143 行执行相同的处理来选出一个二级页表存在的页
    // 目录项，并取得相应二级页表指针。
147                 if (page_entry >= 1024) {
148                         page_entry = 0;
149                 repeat:
150                         dir_entry++;
151                         if (dir_entry >= 1024)
152                                 dir_entry = FIRST_VM_PAGE>>10;
153                         pg_table = pg_dir[dir_entry];  // 页目录项内容。
154                         if (!(pg_table&1))
155                                 if ((counter -= 1024) > 0)
156                                         goto repeat;
157                                 else
158                                         break;
159                         pg_table &= 0xfffff000;         // 页表指针。
160                 }
161                 if (try_to_swap_out(page_entry + (unsigned long *) pg_table))
162                         return 1;
163         }
164         printk("Out of swap-memory\n\r");
165         return 0;
166 }
167
168 /*
169  * Get physical address of first (actually last :-) free page, and mark it
170  * used. If no free pages left, return 0.
171  */
    /*
     * 获取首个(实际上是最后 1 个:-)空闲页面，并标记为已使用。如果没有空闲页面，
     * 就返回 0。
     */
    //// 在主内存区中申请 1 页空闲物理页面。
    // 如果已经没有可用物理内存页面，则调用执行交换处理。然后再次申请页面。
    // 输入：%1(ax=0) - 0；%2(LOW_MEM)内存字节位图管理的起始位置；%3(cx= PAGING_PAGES)；
    // %4(edi=mem_map+PAGING_PAGES-1)。
    // 输出：返回%0（ax = 物理页面起始地址）。函数返回新页面的物理地址。
    // 上面%4 寄存器实际指向 mem_map[]内存字节位图的最后一个字节。本函数从位图末端开始向
    // 前扫描所有页面标志（页面总数为 PAGING_PAGES），若有页面空闲（内存位图字节为 0）则
    // 返回页面地址。注意！本函数只是指出在主内存区的一页空闲物理页面，但并没有映射到某
    // 个进程的地址空间中去。后面的 put_page() 函数即用于把指定页面映射到某个进程的地址
    // 空间中。当然对于内核使用本函数并不需要再使用 put_page() 进行映射，因为内核代码和
    // 数据空间（16MB）已经对等地映射到物理地址空间。
    // 第 65 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中，以便于高效访问和
    // 操作。这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册“在指定寄存
    // 器中的变量”。
```

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

```
216         }
    // 交换数据块总数转换成对应可交换页面总数。该值不能大于 SWAP_BITS 所能表示的页面数。
    // 即交换页面总数不得大于 32768。  然后申请一页物理内存用来存放交换页面位映射数组
    // swap_bitmap，其中每 1 比特代表 1 页交换页面。
217         swap_size >>= 2;
218         if (swap_size > SWAP_BITS)
219                 swap_size = SWAP_BITS;
220         swap_bitmap = (char *) get_free_page();
221         if (!swap_bitmap) {
222                 printk("Unable to start swapping: out of memory :-)\n\r");
223                 return;
224         }
    // read_swap_page(nr, buffer)被定义为 ll_rw_page(READ,SWAP_DEV,(nr),(buffer))。
    // 参见 linux/mm.h 文件第 11 行。这里把交换设备上的页面 0 读到 swap_bitmap 页面中。
    // 该页面是交换区管理页面。其中第 4086 字节开始处含有 10 个字符的交换设备特征字
    // 符串“SWAP-SPACE”。若没有找到该特征字符串，则说明不是一个有效的交换设备。
    // 于是显示信息，释放刚申请的物理页面并退出函数。否则将特征字符串字节清零。
225         read_swap_page(0,swap_bitmap);
226         if (strncmp("SWAP-SPACE",swap_bitmap+4086,10)) {
227                 printk("Unable to find swap-space signature\n\r");
228                 free_page((long) swap_bitmap);
229                 swap_bitmap = NULL;
230                 return;
231         }
232         memset(swap_bitmap+4086,0,10);
    // 然后检查读入的交换位映射图。应该 32768 个比特位全为 0，若位图中有置位的比特位 0，
    // 则表示位图有问题，于是显示出错信息、释放位图占用的页面并退出函数。为了加快检查
    // 速度，这里首先仅挑选查看位图中位 0 和最后一个交换页面对应的比特位，即 swap_size
    // 交换页面对应的比特位，以及随后到 SWAP_BITS（32768）比特位。
233         for (i = 0 ; i < SWAP_BITS ; i++) {
234                 if (i == 1)
235                         i = swap_size;
236                 if (bit(swap_bitmap,i)) {
237                         printk("Bad swap-space bit-map\n\r");
238                         free_page((long) swap_bitmap);
239                         swap_bitmap = NULL;
240                         return;
241                 }
242         }
    // 然后再仔细地检测位 1 到位 swap_size 所有比特位是否为 0。若有不是 0 的比特位存在，
    // 则表示位图有问题，于是释放位图占用的页面并退出函数。否则显示交换设备工作正常
    // 以及交换页面数和交换空间总字节数。
243         j = 0;
244         for (i = 1 ; i < swap_size ; i++)
245                 if (bit(swap_bitmap,i))
246                         j++;
247         if (!j) {
248                 free_page((long) swap_bitmap);
249                 swap_bitmap = NULL;
250                 return;
251         }
252         printk("Swap device ok: %d pages (%d bytes) swap-space\n\r",j,j*4096);
253 }
```