Linux 0.01 内核 mm 模块解读(讨论稿)

Tm-linux 工作组(http://www.tm-linux.com/)
coolday (Email: yuebinqi@163.com)
Wlinux（Email: wlinux@eyou.com）

一、80386 分页机制介绍

这里只针对 memory 相关的一些寄存器作一些解释。80386 处理器的控制寄存器包括了 CR0、CR1、CR2、CR3。其中 CR3 指示页目录表的起始地址。CR0 寄存器的 PE 位（第 0 位）标志分段，该位如为 1 则表处理器工作于保护模式下。PE=0 则表明处理器工作于实地址模式。CR0 的 PG 位（第 31 位）为分页控制位，PG=1 表启用分页机制，PG=0 则禁止分页。

1、 80386 的线性地址

80386 的 32 位线性地址分成 3 个字段，如图 1

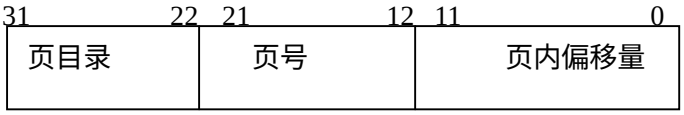

图 1：32 位线性地址

页目录 :为一个大小为 4KB 的内存块,以 4 字节为一目录项,线性地址的高 10 位(22-31)指示页目录地址，其值范围为 0-1023。

页目录的每个入口点的值描述了一个页表，页表和页目录一样，也有 1024 个入口点。32 位线性地址的第二段共 10 位，分别对应于页表的一个入口点。

页表的每个入口点的值表代表一个大小为 4KB 的内存块，叫做页帧，其长度就是页面长度。32 位线性地址的最低段共 12 位，其值范围 0-4095，对应于页帧的每一字节。页帧中存放真正的代码或数据。

80386 页表或页目录中每个表项的格式，如图 2

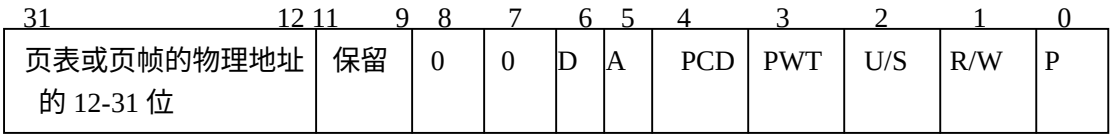

图 2：页目录项和页表项的格式

各位含义：

P：存在位，为 1 则地址转换有效，为 0 由软件自行解释，linux0.01 中表无效

R/W：读写位，为 0 表可读、可执行但不可写，为 1 则可写、可读、可执行，在处理器处于特权级 0、1、2 时，忽略该位

U/S：用户/系统位，为 1 表该页可在任何特权级下访问，为 0 则只能在特权级 0、1、2 下访

问。
A：访问位，在对页进行访问之前，对该位置 1
D：已写标识位，对该页进行写访问之前，该位置 1

2、线性地址到物理地址的转换

这里只标出基于分页的线性地址到物理地址的转换，对于线性的地址的形成，请参考分段机制。

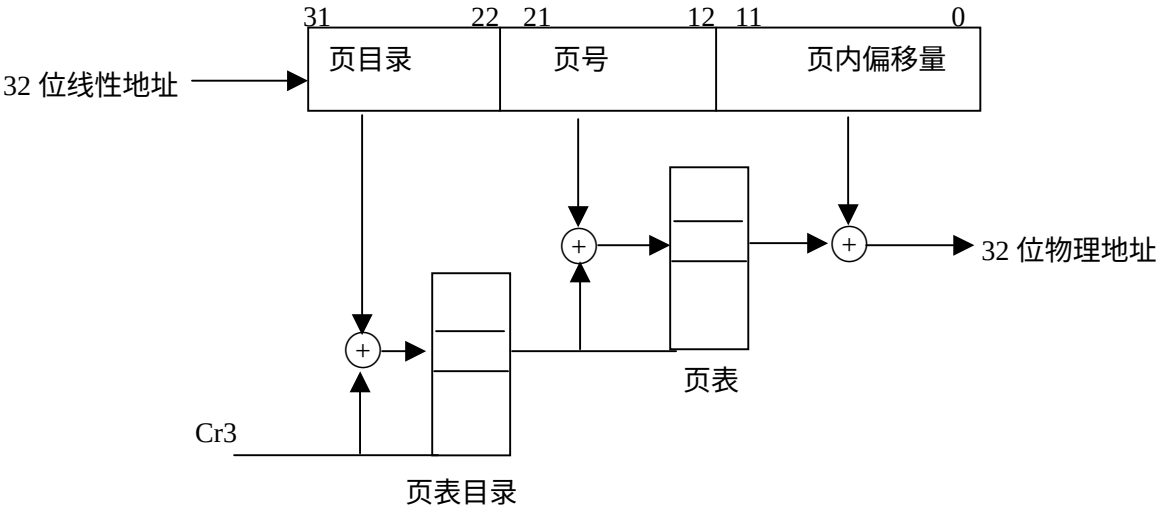

图 3：物理地址的形成

二、linux0.01 中的 mm 管理

其内存管理定义了两种模式，LINUS_HD 和 LASU_HD，不同模式的内存寻址空间不同，两者的定义比较见表 4：

定义项 / 模式	最高地址 HIGH_MEM	缓冲区 BUFF_END	根设备号 ROOT_DEV	硬盘类型 HD_TYPE	内存低位 LOW_M
LINUS_HD	0x800000	0x200000	0x306	{ 5,17,980,300,980,0 }, { 5,17,980,300,980,0 }	0x200000
LASU_HD	0x400000	0xA0000	0x302	{ 7,35,915,65536,920,0 }	0x100000

表 4：linux0.01 支持的两种内存模式

在其代码中采用的是 LINUS_HD，所以以后都以这种为例说明。

按 4KB 分页，每页为 4096 字节。

需要说明的是，linux0.01 的内核内存空间(0~LOW_MEM)并没有进行分页，有内核直接管理。内存分页范围为 LOW_MEM~~HIGH_MEM。

三、mm 相关的数据结构定义

1、 内存映射表：

定义为　static unsigned short mem_map[PAGING_PAGES]

该表中保存的是每个真实页的使用情况，每个表项 16 位，如表项值为 0，则说明该页空闲；如值>0，则说明该页有多少个用户（进程）在使用。

下面的代码中经常涉及到页内存映射表偏移量与页物理地址的转换。由于地址小于 LOW_MEM（0x200000）的内存单元是不列入可分配页中，故 mem_map 映射表的起始页应

该是 LOW_MEM，故映射表中的偏移与实际物理地址存在 LOW_MEM 的偏移。

假设某页物理地址为 addr ，其映射表中偏移为 map_offset，则该页的使用情况由 mem_map[map_offset]标记，addr 与 map_offset 的相互转换如下：

1） addr->map_offset 转换： map_offset=(addr-LOW_MEM)>>12

2） map_offset->addr 转换： addr=map_offset<<12+LOW_MEM

其中左移或右移 12 位是因为 1 页对应 4KB（12 位）

四、mm 模块代码注释

1、memory.c

```
#include <signal.h>
#include <linux/config.h>
#include <linux/head.h>
#include <linux/kernel.h>
#include <asm/system.h>
int do_exit(long code);

#define invalidate() \
__asm__("movl %%eax,%%cr3"::"a" (0))    /*将 cr3 寄存器置 0，即页目录表存放在第 0 页*/

#if (BUFFER_END < 0x100000)
#define LOW_MEM 0x100000
#else    /*限定最低可访问内存，LOW_MEM 以下为内核保护区域，用户进程不能访问*/
#define LOW_MEM BUFFER_END
#endif
/*我们以 LINUX_HD 为例，则 LOW_MEM=BUFF_END=0x200000，以下均为 LINUX_HD
   模式 */
/* these are not to be changed - thay are calculated from the above */
#define PAGING_MEMORY (HIGH_MEMORY - LOW_MEM)
/*可分配内存空间大小 0x600000,即 6MB    */
#define PAGING_PAGES (PAGING_MEMORY/4096)
/*总共页数      0x600，即 1536 页  */
#define MAP_NR(addr) (((addr)-LOW_MEM)>>12)    //求 addr 在 mem_map 表中的偏移量

#if (PAGING_PAGES < 10)
#error "Won't work"
#endif
/*  本函数是将从 from 描述的页（其地址应为物理地址）中的数据拷到 to 描述的页中（同
*  地址应为物理地址），其拷贝字节是 4096 个，其中 movsl 一次拷贝长字（4 字节）
*/
#define copy_page(from,to) \
__asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si")

/*定义页映射表，因是 static 数组，所以所有表项值自动初始化为 0*/
static unsigned short mem_map [ PAGING_PAGES ] = {0,};
```

```
/*
 *  Get physical address of first (actually last :-) free page, and mark it
 * used. If no free pages left, return 0.
 *get_free_page
 *入口：无          出口：32 位物理地址
 * 查找物理地址的第一个空闲页面（实际上地址最高），并且把它标记为被使用
 * 如没有空闲页面，返回值 0。
 */
unsigned long get_f ree_page(void)
{
register unsigned long __res asm("ax");

__asm__("std ; repne ; scasw\n\t"
        "jne 1f\n\t"
        "movw $1,2(%%edi)\n\t"
        "sall $12,%%ecx\n\t"
        "movl %%ecx,%%edx\n\t"
        "addl %2,%%edx\n\t"
        "movl $1024,%%ecx\n\t"
        "leal 4092(%%edx),%%edi\n\t"
        "rep ; stosl\n\t"
        "movl %%edx,%%eax\n"
        "1:"
        :"=a" (__res)
        :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
        "D" (mem_map+PAGING_PAGES-1)
        :"di","cx","dx");
return __res;
}
/*  对于本函数的详细说明：
        :"=a" (__res)
        :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
        "D" (mem_map+PAGING_PAGES-1)
        :"di","cx","dx"
这几条语句的初始化如下：
        ax=0；
        cx=PAGING_PAGES      即 0x600
        di= mem_map+PAGING_PAGES-1    di 值为内存映射表中最高表项的地址
        "std ; repne ; scasw\n\t"    这几条语句完成的是查找空闲页表在内存映射表中的偏移
        di 单元内容与 ax（=0）比较，如果相等则跳到标号“ 1 ”代码段，否则 di-2，继续比较，
        直到 cx 计数器为 0，此时表明所有页表都已使用。
        "jne 1f\n\t"
        "movw $1,2(%%edi)\n\t"       将该空闲页对应的内存映射表项置 1，表使用者为 1
```

```
        "sall $12,%%ecx\n\t"        ecx（空闲页对应的偏移）左移 12 位（4k），转化成物理地址
        "movl %%ecx,%%edx\n\t"      edx=ecx
        "addl %2,%%edx\n\t"         edx=edx+LOW_MEM，edx 值为该页的物理地址
        "movl $1024,%%ecx\n\t"    下面 3 条语句完成的任务是将该页（4092 字节，不包括最
        "leal 4092(%%edx),%%edi\n\t"        低 4 字节(页框内容)）清 0
        "rep ; stosl\n\t"
        "movl %%edx,%%eax\n"      将该页物理地址赋给 eax 返回
*/
/*
 * Free a page of memory at physical address 'addr'. Used by
 * 'free_page_tables()'
 *free_page
 * 释放物理地址是'addr'的一个页面的内存。被使用'free_page_tables()'调用
 * 入口：物理地址'addr'
 * 出口：无
 * 功能及描述:
 *        释放 addr 所在物理页引用，实际操作仅仅将其对应的内存映射表中引用值减一
 *        1。检查地址的合理性，包括与 LOW_MEM、HIGH_MEMORY 比较；
 *        2。找到所属的页面映射表；
 *        3。释放页面，其中如果 mem_map[addr]值为 0，表示没有被引用，则报错；
 *                                否则引用值减一。
 */
void free_page(unsigned long addr)
{
      if (addr<LOW_MEM) return;
      if (addr>HIGH_MEMORY)
            panic("trying to free nonexistent page");
      addr -= LOW_MEM;
      addr >>= 12;
      if (mem_map[addr]--) return;
      mem_map[addr]=0;              //如该页原为空闲，则恢复原值 0
      panic("trying to free free page");
}

/*
 * This function frees a continuos block of page tables, as needed
 * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
 * 释放连续内存页，在'exit()'函数中调用。
 * 如同 copy_page_tables()，这个函数处理单位为 4Mb。
 * 入口：from,size（起始地址，释放内存大小）
 * 出口：整型，0 为正常退出
 * 功能及描述：
 *        以 4MB 为单位释放连续内存块
 *注： 地址从 0 到 4K 的物理地址是页目录空间
```

```
 */
int free_page_tables(unsigned long from,unsigned long size)
{
	unsigned long *pg_table;
	unsigned long * dir, nr;

	if (from & 0x3fffff)			//开始地址应 4MB 对齐
		panic("free_page_tables called with wrong alignment");
	if (!from)			//from=0 内存部分为交换内存，不允许释放
		panic("Trying to free up swapper memory space");
	size = (size + 0x3fffff) >> 22;	//将释放内存大小按 4MB 对齐
 /* 下面一句为计算释放起始地址的页目录偏移量，这里直接计算出了在内存中的实际位置
（因页目录表(CR3)地址为 0），分两步可能更容易理解
	   dir=from>>22;	此句得到地址的页目录表中偏移量
	   dir=dir*4+cr3	此时计算目录项的实际地址，其中 cr3=0，每个表项占 4 个字节
	下面的做法是等效的	*/
	dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
	for ( ; size-->0 ; dir++) {
		if (!(1 & *dir))		//如果对应目录项值的地址无效，即该目录项并未分配，则忽略
			continue;
		pg_table = (unsigned long *) (0xfffff000 & *dir);	//得到页表的物理地址
		for (nr=0 ; nr<1024 ; nr++) {	//对页表中的所有表项进行释放
			if (1 & *pg_table)	//如果该页地址有效，则调用 free_page 释放
				free_page(0xfffff000 & *pg_table);
			*pg_table = 0;	//页表值置 0，空闲
			pg_table++;
		}
		free_page(0xfffff000 & *dir);	//释放对应的页表内存
		*dir = 0;	//该目录表项置 0，表空闲
	}
	invalidate();
	return 0;
}

/*
 *  Well, here is one of the most complicated functions in mm. It
 * copies a range of linerar addresses by copying only the pages.
 * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
 *
 * Note! We don't copy just any chunks of memory - addresses have to
 * be divisible by 4Mb (one page-directory entry), as this makes the
 * function easier. It's used only by fork anyway.
 *
 * NOTE 2!! When from==0 we are copying kernel space for the first
```

```
 * fork(). Then we DONT want to copy a full page-directory entry, as
 * that would lead to some serious memory waste - we just copy the
 * first 160 pages - 640kB. Even that is more than we need, but it
 * doesn't take any more memory - we don't copy-on-write in the low
 * 1 Mb-range, so the pages can be shared with the kernel. Thus the
 * special case for nr=xxxx.
 *  这里有一个在 mm 中最复杂的函数。它通过仅仅拷贝页表实现拷贝一段线性地址内容。
 *
 *注意！我们没有拷贝任何块内存——地址必须可以被 4MB 划分（一个目录页），
 *这样做函数实现更简单。且仅仅在 fork 中被使用。
 *
 *注意 2！！如果 from==0 我们为第一个 fork()拷贝内核空间。然后我们不希望
 *拷贝整个目录页，这样做将导致内存的严重浪费——我们仅仅拷贝最开始的 160 页——
 *也就是 640KB。甚至这些都比我们的需要多，不需要更多的内存——我们
 *没有在低于 1Mb 的范围内使用 copy-on-write 机制，故这些页面可以与内核共享。
 *特殊的例子是 nr=xxxx。
 *
 *入口：from,to,size（源地址，目的地址，大小）
 *出口：int,0 为成功，-1 为没有可分配空间
 *功能和描述：
 *        完成页拷贝功能，但并非直接拷贝数据，而是通过拷贝页表来完成的
 *注： get_free_page 的此处作用是得到一个 4KB 的页面，供页目录的页表存放使
*用而不是供存储页数据。
```

拷贝页表数据的结果是被拷贝的页由源和目标共享,见下图：

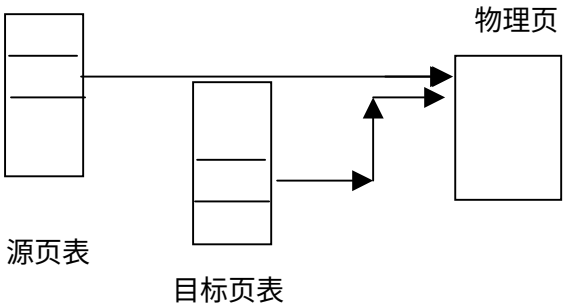

图 5：页拷贝示意图

```
*/
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
        unsigned long * from_page_table;
        unsigned long * to_page_table;
        unsigned long this_page;
        unsigned long * from_dir, * to_dir;
        unsigned long nr;

        if ((from&0x3fffff) || (to&0x3fffff))      //源和目标地址的起始地址都应 4MB 对齐
```

```
            panic("copy_page_tables called with wrong alignment");
 /*    from_dir ,to_dir  分别计算源和目标对应目录表项地址  */
        from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
        to_dir = (unsigned long *) ((to>>20) & 0xffc);
        size = ((unsigned) (size+0x3fffff)) >> 22;      //拷贝字节 4MB 对齐
        for( ; size-->0 ; from_dir++,to_dir++) {
            if (1 & *to_dir)    //如果 to_dir 对应目录表项已使用，则出错处理
                panic("copy_page_tables: already exist");
            if (!(1 & *from_dir))    //如果 from_dir 对应目录表项空闲，则跳过
                continue;
/*  得到源地址对应页表的物理地址    */
            from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
/*注意：这里创建一个新的页表，以存放拷贝的页表项    */
            if (!(to_page_table = (unsigned long *) get_free_page()))
                return -1; /* Out of memory, see freeing */
/*将目标对应的目录表项指向刚申请到的页表，并将目录表项置为地址有效、可读写、用户
   特权级  */
            *to_dir = ((unsigned long) to_page_table) | 7;
/*确定需拷贝的页表项数目，如果 from=0,则为页目录表的拷贝，只需 0xa0，否则为 1024*/
            nr = (from==0)?0xA0:1024;
            for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
                this_page = *from_page_table;
                if (!(1 & this_page))      //如果拷贝源的当前页地址无效，则跳过
                    continue;
                this_page &= ~2;         //将该页表项设为只读
                *to_page_table = this_page;    //将页表项值赋给目标页表项
                if (this_page > LOW_MEM) {
                    *from_page_table = this_page;    //将拷贝源取消写权利
                    this_page -= LOW_MEM;
                    this_page >>= 12;
                    mem_map[this_page]++;      //将该表在页映射表中对应项的引用值加 1
                }
            }
        }
        invalidate();
        return 0;
}

/*
 * This function puts a page in memory at the wanted address.
 * It returns the physical address of the page gotten, 0 if
 * out of memory (either when trying to access page-table or
 * page.)
```

```
 * 0 表示内存分配完（不管是在要得到访问的页面表还是页面的时候）
 *
 *入口：page,address（物理空间地址，线性空间地址）
 *出口：unsigned long（0 为没有空间，其他为入口的 page 物理地址）
 *功能和描述：
 *        建立物理空间和线性地址空间的页面对应关系
 *        1。检查
 *        2。设置 page 相应的页目录和页表，并将 addr 对应的页表项指向该页
 *        3。返回该页物理地址，若分配页出错返回 0
 */
unsigned long put_page(unsigned long page,unsigned long address)
{
    unsigned long tmp, *page_table;

/* NOTE !!! This uses the fact that _pg_dir=0 */
    if (page < LOW_MEM || page > HIGH_MEMORY)
        printk("Trying to put page %p at %p\n",page,address);
/*如果 page 在页映射表对应项并非一人独享或地址无效，则出错处理 */
if (mem_map[(page-LOW_MEM)>>12] != 1)
        printk("mem_map disagrees with %p at %p\n",page,address);
    page_table = (unsigned long *) ((address>>20) & 0xffc);
    if ((*page_table)&1)  //如果目标地址对应的目录表项地址有效，则读取对应的页表地址
        page_table = (unsigned long *) (0xfffff000 & *page_table);
    else {   //否则申请一个新的页表，将该目录表项指向该页表
        if (!(tmp=get_free_page()))
            return 0;
        *page_table = tmp|7;   //将目录表项设为地址有效、可读写、用户特权级
        page_table = (unsigned long *) tmp;
    }
 /*将 addr 对应的页表项值指向该物理页，(addr>>12)&0x3ff 为计算页表中偏移量 */
    page_table[(address>>12) & 0x3ff] = page | 7;
    return page;
}

/*un_wp_page
 *入口：table_entry（页表指针）
 *出口：无
 *功能及描述：
 *        将页表指向的页帧的读写性设为可写（若有多个引用则拷贝该页后设为可写）
 *        1。找到页表指向的页帧的物理地址
 *        2。如该页地址不在内核空间并且相应的物理页面引用值为 1 时，直接设为可写
 *        3。否则复制该页，并设为可写（copy-on-write 机制）
 */
void un_wp_page(unsigned long * table_entry)
```

```
{
	unsigned long old_page,new_page;
	old_page = 0xfffff000 & *table_entry;	//得到页表项对应的页物理地址
/*如果该页地址不在保留内存区且使用者只有 1 个 ，则将其页表项改为可写，返回*/
	if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
		*table_entry |= 2;
		return;
	}
/*否则申请一个空闲页，将原页拷到新页中，并将该使用者的页表项指向新页，新页可读写
*/
	if (!(new_page=get_free_page()))
		do_exit(SIGSEGV);
	if (old_page >= LOW_MEM)
		mem_map[MAP_NR(old_page)]--;
	*table_entry = new_page | 7;
	copy_page(old_page,new_page);
}

/* do_wp_page
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
 *入口：error_code,address（error_code 未使用，address 线性地址）
 *出口：无
 *功能及描述：
 *		当使用者试图写一个共享页的时候调用，实际工作通过 un_wp_page 完成
 */
void do_wp_page(unsigned long error_code,unsigned long address)
{   /*这里直接计算线性地址 address 对应的页表项地址
       其中(address>>20) &0xffc)得到目录表项地址，0xfffff000 &则得到页表物理地址，
        (address>>10) & 0xffc)则得到页号，最后相加得到页表项地址 */
	un_wp_page((unsigned long *)
		(((address>>10) & 0xffc) + (0xfffff000 &
		*((unsigned long *) ((address>>20) &0xffc)))));

}
/*write_verity
 *入口：address（线性地址）
 *出口：无
 *功能及描述：
 *		确认线性地址 address 的对应页面的可写性（不可写的时候则复制并设为可写）
 *		1。测试页目录的有效性，并得到页目录内容
 *		2。得到 address 的页表的物理地址
 *		3。如果有效不可写则复制物理页面并设置可写（调用 un_wp_page）
```

```
 */
void write_verify(unsigned long address)
{
      unsigned long page;
   /*如果对应页目录表项地址无效，则返回 */
      if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
            return;
      page &= 0xfffff000;   //得到页表地址
      page += ((address>>10) & 0xffc);   //得到 address 的对应页表项地址
   /*如果页表项对应页地址有效但不可写，则调用 un_wp_page 函数进行可写标记 */
      if ((3 & *(unsigned long *) page) == 1)    /* non-writeable, present */
            un_wp_page((unsigned long *) page);
      return;
}
/*do_no_page
 *入口：error_code,address（error_code 出错代码未使用，address 线性地址）
 *出口：无
 *功能及描述：
  *    处理某任务页不足的请求，函数申请一个空闲页，并放入对应的地址，然后发送信号
  *    量
 */
void do_no_page(unsigned long error_code,unsigned long address)
{
      unsigned long tmp;

      if (tmp=get_free_page())
            if (put_page(tmp,address))
                  return;
      do_exit(SIGSEGV);
}
/*calc_mem
 *入口：无
 *出口：无
 *功能及描述：
 *        打印两类信息：空闲的物理内存页面数目
 *                      线性空间的各个有效页目录中有效页表的数目（0，1 页目录除外）
 */
void calc_mem(void)
{
     int i,j,k,free=0;
     long * pg_tbl;
/* 以下计算空闲的页数目 */
      for(i=0 ; i<PAGING_PAGES ; i++)
            if (!mem_map[i]) free++;      //mem_map[i]=0,则该页空闲
```

```
        printk("%d pages free (of %d)\n\r",free,PAGING_PAGES);
/*计算每个页目录表项（不包括第 0、1 个目录表项）的使用的页数 */
for(i=2 ; i<1024 ; i++) {
            if (1&pg_dir[i]) {
                pg_tbl=(long *) (0xfffff000 & pg_dir[i]);
                for(j=k=0 ; j<1024 ; j++)
                    if (pg_tbl[j]&1)   //如页表项地址有效，则使用了某页
                        k++;
                printk("Pg-dir[%d] uses %d pages\n",i,k);
            }
        }
}
```

2、 page.s

```
/*
 * page.s contains the low-level page-exception code.
 * the real work is done in mm.c
   page.s 为页异常的处理，目前只处理了地址无效和共享页面写导致的错误
 */
.globl _page_fault

_page_fault:
    xchgl %eax,(%esp)
    pushl %ecx
    pushl %edx
    push %ds
    push %es
    push %fs    //以上为寄存器压栈，以便返回继续执行
    movl $0x10,%edx
    mov %dx,%ds
    mov %dx,%es
    mov %dx,%fs  //以上为重置各寄存器值
    movl %cr2,%edx  //将cr2保存的出错地址移入edx
    pushl %edx  //传入两个参数：1）eax:出错页表描述项
    pushl %eax  //              2）edx：出错页地址
    testl $1,%eax  //如果出错标志（第0位）为0，则为写出错，调用do_wp_page处理
    jne 1f
    call _do_no_page  //否则无空页，调用do_no_page申请空闲页
    jmp 2f
1:  call _do_wp_page
2:  addl $8,%esp    //忽略压入的edx,eax参数
    pop %fs        //以下弹出压栈寄存器，返回
    pop %es
    pop %ds
    popl %edx
```

```
popl %ecx
popl %eax
iret
```