Linux0.11 下的内存管理

作者：袁镱

robertyi@163.com

QQ:30131195

愿借此结交广大 linux 爱好者

1 如何在保护模式下实现对物理内存的管理

保护模式在硬件上为实现虚拟存储创造了条件，但是内存的管理还是要由软件来做。操作系统作为资源的管理者，当然要对内存的管理就要由他来做了。

在 386 保护模式下，对任何一个物理地址的访问都要通过页目录表和页表的映射机制来间接访问，而程序提供的任何地址信息都会被当成线性地址进行映射，这就使得地址提供者不知道他所提供的线性地址最后被映射到了哪个具体的物理地址单元。这样的措施使得用户程序不能随意地操作物理内存，提高了系统的安全性，但是也给操作系统管理物理内存造成了障碍。而操作系统必须要了解物理内存的使用情况才谈得上管理。

要能够在保护模式下感知物理内存，也就是说要能够避开保护模式下线性地址的影响，直接对物理内存进行操作。如何避开呢？正如前面所说：在保护模式下对任何一个物理地址的访问都要通过对线性地址的映射来实现。

不可能绕过这个映射机制，那只有让他对内核失效。如果让内核使用的线性地址和物理地址重合，比如：当内核使用 0x0000 1000 这个线性地址时访问到的就是物理内存中的 0x00001000 单元。问题不就解决了吗！linux0.11 中采用的正是这种方法。

在进入保护模式之前，要初始化页目录表和页表，以供在切换到保护模式之后使用，要实现内核线性地址和物理地址的重合，必须要在这个时候在页目录表和页表上做文章。

在看代码之前首先说明几点：

由于 linus 当时编写程序时使用的机器只有 16M 的内存，所以程序中也只处理了 16M 物理内存的情况，而且只考虑了 4G 线性空间的情况。一个页表可以寻址 4M 的物理空间，所以只需要 4 个页表，一个页目录表可以寻址 4G 的线性空间，所以只需要 1 个页目录表。

程序将页目录表放在物理地址_pg_dir＝0x0000 处，4 个页表分别放在 pg0＝0x1000, pg1＝0x2000, pg2＝0x3000, pg3＝0x4000 处

下面是最核心的几行代码：在 linux/boot/head.s 中

首先对 5 页内存清零

```
198 setup_paging:
199 movl $1024*5,%ecx           /* 5 pages - pg_dir+4 page tables */
                                #设置填充次数 ecx=1024*5
200 xorl %eax,%eax              #设置填充到内存单元中的数 eax=0
201 xorl %edi,%edi              /* pg_dir is at 0x000 */
                        #设置填充的起始地址 0，也是页目录表的起始位置
202 cld;rep;stosl
```

下面填写页目录表的页目录项

对于 4 个页目录项，将属性设置为用户可读写，存在于物理内存，所以页目录项的低 12 位是 0000 0000 0111B

以第一个页目录项为例，$ pg0+7=0x0000 1007

表示第一个页表的物理地址是 0x0000 1007&0xffff f000=0x0000 1000;

权限是 0x0000 1007&0x0000 0fff=0x0000 0007

```
203 movl $pg0+7,_pg_dir             /* set present bit/user r/w */
204 movl $pg1+7,_pg_dir+4           /* --------- " " --------- */
205 movl $pg2+7,_pg_dir+8           /* --------- " " --------- */
206 movl $pg3+7,_pg_dir+12          /* --------- " " --------- */
```

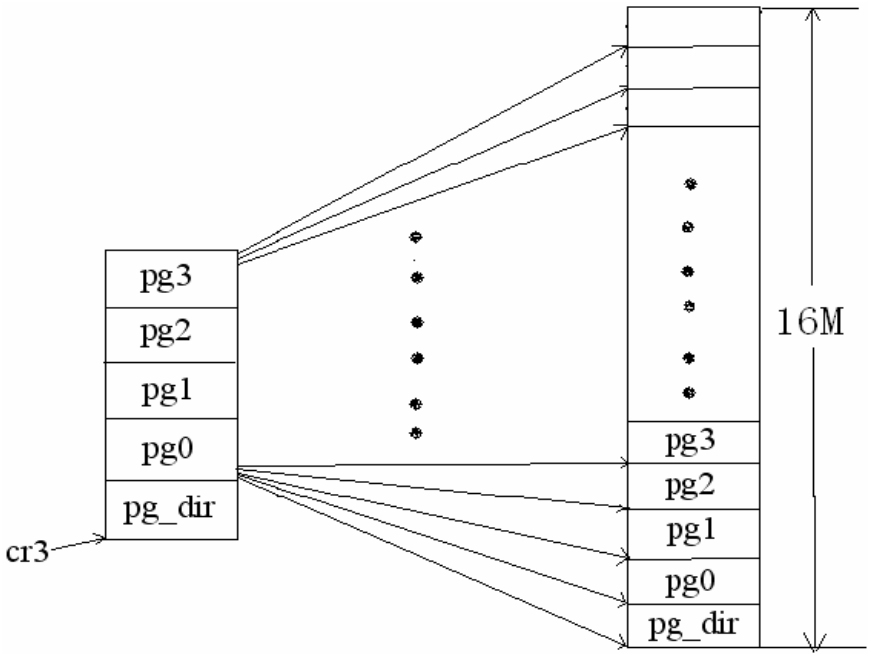
接着便是对页表的设置:

4 个页表×1024 个页表项×每个页表项寻址 4K 物理空间：4*1024*4*1024=16M

每个页表项的内容是：当前项所映射的物理内存地址 + 该页的权限
其中该页的属性仍然是用户可读写，存在于物理内存，即 0x0000 0007
具体的操作是从 16M 物理空间的最后一个页面开始逆序填写页表项：
最后一个页面的起始物理地址是 0x0xfff000，加上权限位便是 0x fff007,以后每减 0x1000（一个页面的大小）便是下一个要填写的页表项的内容。

```
207 movl $pg3+4092,%edi       # edi 指向第四个页表的最后一项 4096-4。
208 movl $0xfff007,%eax       /* 16Mb - 4096 + 7 (r/w user,p) */
                    #把第四个页表的最后一项的内容放进 eax
209 std                       # 置方向位，edi 值以 4 字节的速度递减。
210 1: stosl                  /* fill pages backwards - more efficient :-) */
211 subl $0x1000,%eax         # 每填写好一项，物理地址值减 0x1000。
212 jge 1b                    # 如果 eax 小于 0 则说明全填写好了。
        # 使页目录表基址寄存器 cr3 指向页目录表。
213 xorl %eax,%eax           /* pg_dir is at 0x0000 */
                              令 eax=0x0000 0000(页目录表基址)
214 movl %eax,%cr3           /* cr3 - page directory start */
      # 设置 cr0 的 PG 标志（位 31），启动保护模式
215 movl %cr0,%eax
216 orl $0x80000000,%eax     # 添上 PG 标志位。
217 movl %eax,%cr0           /* set paging (PG) bit */
```

在分析完这段代码之后，应该对初始化后的页目录表和页表有了一个大概的了解了，当这段代码运行完后内存中的映射关系应该如图所示：



接下来将内核代码段描述符 gdt 设置为
0x00c09a0000000fff /* 16Mb */ # 代码段最大长度 16M。
这样线性地址就和物理地址重合了。
下面用两个例子验证一下：
（1） 要寻找 pg_dir 的第 15 项的内容

这个地址应该是在页目录表的(15-1)*4=0x38 位置，把它写成 32 为地址使 0x0000 0038，当内核使用这个地址时，仍然要通过映射：首先取高 10 位，0000 0000 00B，根据 203 行的代码，页目录表第 0 项的内容是$pg0+7，得到页表地址是 pg0=0x0000 1000,CPU 将用这个地址加上偏移量找到对应的页表项，偏移量＝线性地址中间 10 位*4＝0，根据 203～221 行执行的结果，在 pg0 中偏移量为 0 的页表项为 0x0000 0007, CPU 得到页表地址是 0x0000 0000 加上线性地址的最后 12 位，将找到 0x0000 0038 单元的内容。

（2）寻找任意物理单元 0x00f5 9f50

与第一个例子一样，用这个地址作为线性地址寻址，先用高 10 位寻找页表，页目录表第 0000 0000 11B 项指向 pg3，根据线性地址中间 10 位 11 0101 1001B 寻找页表项，pg3 的第 11 0101 1001B 应该是 0x00f5 9007，

取得页表基址 0x00f5 9000，加上页内偏移量 0x f50，最后得到的就是物理地址 0x00f5 9f50 的内容。

从上面两个例子可以看出：内核中使用的线性地址实际上已经是物理地址，这样从现象上看 386 的地址映射机制对内核失效了:-)

明白了这一点之后，对后面内存管理方面的的分析就容易得多了

2 内存初始化

当操作系统启动前期实现对于物理内存感知之后，接下来要做的就是对物理内存的管理，要合理的使用。对于 Linux 这样一个操作系统而言，内存有以下一些使用：面向进程，要分配给进程用于执行所必要的内存空间；面向文件系统，要为文件缓冲机制提供缓冲区，同时也要为虚拟盘机制提供必要的空间。这三种对于内存的使用相对独立，要实现这一些，就决定了物理内存在使用时需要进行划分，而最简单的方式就是分块，将内存划分为不同的块，各个块之间各司其职，互不干扰。linux0.11 中就是这样作的。

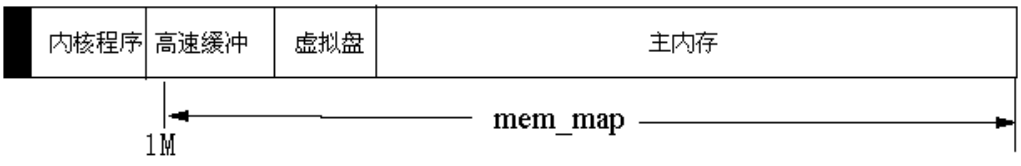
Linux0.11 将内存分为内核程序、高速缓冲、虚拟盘、主内存四个部分（黑色部分是页目录表、几个页表，全局描述符表，局部描述符表。一般将他们看作内核的一部分）。为什么要为内核程序单独划出一个块来呢？主要是为了实现上简单。操作系统作为整个计算机资源的管理者，内核程序起着主要的作用，它的代码在操作系统运行时会经常被调用，需要常驻内存。所以将这部分代码与一般进程所使用的空间区分开，为他们专门化出一块内存区域。专门划出一块区域还有一个好处，对于内核程序来说，对于自己的的管理就简单了，内核不用对自己代码进行管理。比如：当内核要执行一个系统调用时，发现相应的代码没有在内存，就必须调用相关的内核代码去将这个系统调用的代码加载到内存，在这个过程中，有可能出现再次被调用的相关内核代码不在内存中的情况，最后就可能会导致系统崩溃。操作系统为了避免这种情况，在内核的设计上就变得复杂了。如果将内核代码专门划一个块出来，将内核代码全部载入这个块保护起来，就不会出现上面讲的情况了。

在 linux0.11 中内存管理主要是对主内存块的管理。

要实现对于这一块的管理，内核就必须对这一块中的每一个物理页面的状态很清楚。一个物理页面应该有以下基本情况：是否被分配，对于它的存取权限（可读、可写），是否被访问过，是否被写过，被多少个不同对象使用。对于 linux0.11 来说，后面几个情况可以通过物理页面的页表项的 D、A、XW 三项得到，所以对于是否被分配，被多少个对象使用就必须要由内核建立相关数据结构来记录。在 linux0.11 定义了一个字符数组 mem_map [PAGING_PAGES] 用于对主内存区的页面分配和共享信息进行记录。

以下代码均在 /mm/memory.c 中

```
43 #define LOW_MEM        0x100000                    // 主内存块可能的最低端（1MB）。
44 #define PAGING_MEMORY (15*1024*1024)               // 主内存区最多可以占用 15M。
45 #define PAGING_PAGES (PAGING_MEMORY>>12)     // 主内存块最多可以占用的物理页面数
46 #define MAP_NR(addr) (((addr)-LOW_MEM)>>12)    // 将指定物理内存地址映射为映射数组标号。
47 #define USED 100                                   // 页面被占用标志
57 static unsigned char mem_map [ PAGING_PAGES ] = {0,};    // 主内存块映射数组
```



mem_map 中每一项的内容表示物理内存被多少个的对象使用，所以对应项为 0 就表示对应物理内存

页面空闲。

可以看出当内核在定义映射数组 mem_map 时是以主内存块最大可能大小 15M 来定义的，最低起始地址为 LOW_MEM，mem_map 的第一项对应于物理内存的地址为 LOW_MEM，所以就有了第 46 行的映射关系 MAP_NR。而当实际运行时主内存块却不一定是这么大，这就需要根据实际主内存块的大小对 mem_map 的内容进行调整。对于不是属于实际主内存块的物理内存的对应项清除掉，linux0.11 采用的做法是在初始化时将属于实际属于主内存块的物理内存的对应项的值清零，将不属于的置为一个相对较大的值 USED。这样在作管理时这些不属于主内存块的页面就不会通过主内存块的管理程序被分配出去使用了。

下面就是主内存块初始化的代码：

/init/main.c

当系统初启时，启动程序通过 BIOS 调用将 1M 以后的扩展内存大小（KB）读入到内存 0x90002 号单元

```
58 #define EXT_MEM_K    (*(unsigned short *)0x90002)
```

下面是系统初始化函数 main() 中的内容

```
112 memory_end = (1<<20) + (EXT_MEM_K<<10);        // 内存大小 =1Mb 字节 + 扩展内存 (k)*1024 字节。
113 memory_end &= 0xfffff000;                      // 以页面为单位取整。
114 if (memory_end > 16*1024*1024)                 // linux0.11 最大支持 16M 物理内存
115          memory_end = 16*1024*1024;
116 if (memory_end > 12*1024*1024)                 // 根据内存大小设置缓冲区末端的位置
117         buffer_memory_end = 4*1024*1024;
118   else if (memory_end > 6*1024*1024)
119             buffer_memory_end = 2*1024*1024;
120         else
121             buffer_memory_end = 1*1024*1024;
122   main_memory_start = buffer_memory_end;          // 主内存起始位置 = 缓冲区末端；
123 #ifdef RAMDISK                                   // 如果定义了虚拟盘，重新设置主内存块起始位置
                                                    //rs_init() 返回虚拟盘的大小
124     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
125 #endif
126 mem_init(main_memory_start,memory_end);          // 初始化主内存块
```

下面就是 mem_init 的代码。

```
399 void mem_init(long start_mem, long end_mem)
400 {
401     int i;
402
403     HIGH_MEMORY = end_mem;                       // 设置物理内存最高端。
404     for (i=0 ; i<PAGING_PAGES ; i++)             // 将主内存块映射数组所有项置为 USED
405         mem_map[i] = USED;
406     i = MAP_NR(start_mem);                       // 计算实际主内存块物理地址起始位置对应的映射项
407     end_mem  - = start_mem;                      // 计算实际主内存块大小
408     end_mem >>= 12;                              // 计算需要初始化的映射项数目
```

```
409     while (end_mem-->0)                    // 将实际主内存块对应的映射项置为 0（空闲）
410         mem_map[i++]=0;
411 }
```

通过以上的操作之后，操作系统便可以了解主内存块中物理内存页面的使用情况了。

3 内存的分配与回收

分配

当内核本身或者进程需要一页新的物理页面时，内核就要给他分配一个空闲的物理页面。内核需要查询相关信息，以尽量最优的方案分配一个空闲页面，尤其是在有虚存管理机制的操作系统中对于空闲页面的选取方案非常重要，如果选取不当将导致系统抖动。linux0.11 没有实现虚存管理，也就不用考虑这些，只需要考虑如何找出一个空闲页面。

知道了内核对主内存块中空闲物理内存页面的映射结构 mem_map，查找空闲页面的工作就简单了。只需要在 mem_map 找出一个空闲项，并将该项映射为对应的物理页面地址。算法如下：

```
算法：get_free_page
输入：无
输出：空闲页面物理地址
{
      从最后一项开始查找mem_map 空闲项；
      if( 没有空闲项 )
            renturn   0；
      将空闲项内容置 1，表示已经被占用；
      将空闲项对应的下标转换为对应的物理页面的物理地址＝
            ( 数组下标 <<12 ＋ LOW_MEM)
      将该物理页内容清零
      return   对应的物理地址；
}
```

get_free_page 的源码如下：

/mm/memory.c

```
59 /*
60 * Get physical address of first (actually last :-) free page, and mark it
61 * used. If no free pages left, return 0.
62 */
/*
* 获取首个 ( 实际上是最后 1 个 :-) 空闲页面，并标记为已使用。如果没有空闲页面，
* 就返回 0。
*/
// 输入：%1 与 %0 相同表示 eax，初值为 0；%2 表示直接操作数 (LOW_MEM)；
          %3 表示 ecx，初值为 PAGING PAGES；搜索次数
          %4 表示 edi 初值为映射数组最后一项地址 mem_map+PAGING_PAGES-1。
// 输出：返回 %0, 表示 eax 页面起始地址。eax 即 __res
63 unsigned long get_free_page(void)
64 {
65 register unsigned long __res asm( "ax");
66
67     __asm__( "std ; repne ; scasb\n\t"       // 置方向位，将 al(0) 与 (edi) 开始的反相 ecx 个字节的内容比较
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        // 将该内存映射项置 1。
```

```
70                  "sall $12,%%ecx\n\t"            // 相对于 LOW_MEM 的页面起始地址。
71                  "addl %2,%%ecx\n\t"             // 加上 LOW_MEM ＝ > 页面实际物理起始地址。
72                  "movl %%ecx,%%edx\n\t"          // 保存页面实际物理起始地址。
73                  "movl $1024,%%ecx\n\t"          // 置计数值 1024
74                  "leal 4092(%%edx),%%edi\n\t"    // 使 edi 指向该物理页末端
75                  "rep ; stosl\n\t"               // 沿反方向将该页清零。
76                  "movl %%edx,%%eax\n"            // 将页面实际物理起始地址放入 eax（返回值）。
77                  "1:"
78                  : "=a" (__res)
79                  : "" (0), "i" (LOW_MEM), "c" (PAGING_PAGES),
80                  "D" (mem_map+PAGING_PAGES-1)
81                  : "di", "cx", "dx");
82              return __res;                       // 返回空闲页面实际物理起始地址（如果无空闲也则返回 0）。
83 }
84
```

这个函数返回的只是物理页面的物理地址，下一节将具体讲如何将物理地址映射为线性地址。

回收：

当内核使用完一个物理页面或者进程退出时内核归还申请了的物理页面。这时就需要更改相应的信息，以便下一次使用。在归还页面时可能会出现下面几种情况：

1）页面物理地址低于主内存块可能的最低端，这种情况不需要处理直接退出，因为这部分内存空间被用于内核程序和缓冲，没有作为分配页面的内存空间。还有一种情况会出现这种情况，当内存操作失败时，会调用回收页面过程回收已经分配了的物理页，如果因为内存分配失败造成的，就不需要真正的回收操作，调用回收过程时会以 0 为输入参数。

2）页面物理地址高于实际物理内存最高地址。这种情况是不允许的，内核将使调用对象进入死循环，这是一种简单而有效的方法，因为这种情况要判断出错原因是很困难的。

3）调用对象试图释放一块空闲物理内存。出现这种情况可能是因为多个对象共享该物理页，在释放时出现了重复释放。比如：进程 A、B 共享物理页 170，由于系统的原因 A 将该页释放了两次，当 B 释放该页时就会出现这种情况。这种情况也是不允许的，一般意味着内核出错，内核将使调用对象进入死循环以避免错误扩散。

4）要释放的页面正确。因为可能是共享内存，所以要将该页对应的映射项的值减 1，表示减少了一个引用对象。如果引用数减到 0 了，并不对物理页面的内容清 0，等到被分配时再做，因为可能这个页面不会在被使用，同时在分配时用汇编代码来做效率会很高。

这样下面的代码就很好理解了：

```
85 /*
86 * Free a page of memory at physical address 'addr'. Used by
87 * 'free_page_tables()'
88 */
/*
* 释放物理地址 'addr' 开始的一页内存。用于函数 'free_page_tables()'。
```

```
*/
89 void free_page(unsigned long addr)
90 {
91     if (addr < LOW_MEM) return;        // 如果物理地址 addr 小于主内存块可能的最低端，则返回。
92     if (addr >= HIGH_MEMORY)
                                          // 如果物理地址 addr>= 实际内存大小，则显示出错信息，调用对象死机。
93             panic( "trying to free nonexistent page");
94     addr - = LOW_MEM;                          // 将物理地址换算为对应的内存映射数组下标。
95     addr >>= 12;
96     if (mem_map[addr]--) return;               // 如果对应内存映射数组项不等于 0，则减 1，返回
97     mem_map[addr]=0;                         // 否则置对应映射项为 0，并显示出错信息，调用对象死机。
98         panic( "trying to free free page");
99 }
100
```

4 页面映射

如果进程请求一页空闲内存，或者页失效错误时，会出现页面请求。在这个时候请求是以线性地址的形式提出来的。因为对于一个进程来说，它感知不到其他进程的存在，对它自己，觉得独占了所有资源。操作系统在控制物理内存的同时又要控制进程的虚拟空间，这就需要在内存线性地址与物理地址之间作转换工作。比如：进程在线性地址 0x0104 F380 处 产生了缺页中断，内核将进行一系列的处理，最后分配一个物理页面，但是并不能这样返回进程执行，因为进程仍然需要从线性地址 0x0104 F380 处读取数据，就像没有发生过缺页中断一样。操作系统就必须要做这个工作，将物理页面映射到线性地址上。

要将物理页面映射到线性地址上，就应该修改页目录表和页表的相关内容，这样进程才能通过线性地址找到相应的物理页面。回顾一下 386 页面映射机制，cpu 通过线性地址的高 10 位寻找到相应的页表，再通过中间 10 位寻找到物理页面，最后通过低 12 位在物理页面中寻找到相应的内存单元。所以要让进程找到物理页面，就必须根据线性地址设置页目录项和页表项。linux0.11 使用 put_page 来作这个处理，其算法如下：

```
算法：put_page
输入：物理页面地址 page
        线性地址 address
输出：如果成功，返回 page；如果失败，返回 0
{
        if ( 物理页面地址低于 LOW_MEM 或者不小于 HIGH_MEMORY)
              显示出错信息，返回 0；
        if ( 物理页面地址对应的内存映射数组映射项的值！＝ 1)
              显示出错信息，返回 0；
        根据线性地址高 10 位找到对应的页目录表项；
        if ( 页目录表项对应的页表在内存中 )
              根据页目录表项的到页表的物理地址；
           else{
              分配新的物理页面作为新的页表；
              初始化页目录表项，使它指向新的页表；
              根据页目录表项的到页表的物理地址；
               }
        根据线性地址中间 10 位找到对应的页表项；
        if( 对应的页表项已经被使用 )
             显示出错信息，返回 0；
         设置对应的页表项，使它指向物理页面；
        return 物理页面地址；
 }
```

put_page 操纵的是由 get_free_page（）分配得到的物理页面，所以物理页面地址应该是在主内存块中，如果不在，就应该终止映射，返回失败。然后调用 put_page 函数的对象根据自身的特性作相关处理。同样是因为 put_page 操纵的是新分配的物理页面，所以物理页面地址对应的内存映射数组映射项的值应该是 1。如果不是 1，也应该终止映射，返回失败。如果前面的检查通过了，就应改进行映射了。首先在页目录表中找到对应页目录项，如果页目录项有效，即对应页表在内存中，就直接寻找页表项。否则就必须先分配一个物理页作为页表。从理论上讲，在设置对应的页表项之前应该检查一下该页表项是否已经被使用。从而确保映射的一致性，因为如果页表项已经被使用，对其的第二次赋值会使原来的映射关系失效。但是由于 linux 在总体设计上的特点，而且新分配的页表被全部清零，所以不会出现这个问题。随着对代码分析的深入，将体会到这一点。

下面就是 put_page 的代码：

/mm/memory.c

```
190
191 /*
192 * This function puts a page in memory at the wanted address.
193 * It returns the physical address of the page gotten, 0 if
194 * out of memory (either when trying to access page-table or
195 * page.)
196 */
/*
* 下面函数将一内存页面放置在指定地址处。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。
*/
197 unsigned long put_page(unsigned long page,unsigned long address)
198 {
199         unsigned long tmp, *page_table;
200
201 /* NOTE !!! This uses the fact that _pg_dir=0 */
    /* 注意 !!! 这里使用了页目录基址 _pg_dir=0 的条件 */
202
203     if (page < LOW_MEM || page >= HIGH_MEMORY)        // 判断是否在主内存块中
204             printk( "Trying to put page %p at %p\n",page,address);
205     if (mem_map[(page-LOW_MEM)>>12] != 1)                  // 判断对应映射项的值是否为 1
206             printk( "mem_map disagrees with %p at %p\n",page,address);
207     page_table = (unsigned long *) ((address>>20) & 0xffc);   // 根据线性地址找到对应的页目录表项；
208     if ((*page_table)&1)                                                  // 判断页表是否存在
209         page_table = (unsigned long *) (0xfffff000 & *page_table);        // 取对应页表物理地址
210       else {
211             if (!(tmp=get_free_page()))                                  // 申请新物理页作为页表
212                     return 0;
213             *page_table = tmp|7;                                        // 设置页目录项
214             page_table = (unsigned long *) tmp;
215             }
216     page_table[(address>>12) & 0x3ff] = page | 7;                 // 页面设置为用户权限、可写、有效
217 /* no need for invalidate */
/* 不需要刷新页变换高速缓冲 */
218     return page;                                          // 返回物理页面地址。
219 }
220
```

在这个代码中，如果第一个判断为真时，只是打印出错信息，并没有返回。这将导致第二个判断时

mem_map 数组溢出，由于 c 语言并不对数组溢出进行出错处理。这里将可能出现错误。而且第二个判断也没有在打印错误信息之后返回，这将导致错误蔓延。不过幸运的是，linux0.11 中不会以这种参数调用 put_page，所以这里只是作一个算法上的说明。

看了 put_page 之后，那么 get_empty_page 的代码就很好理解了。get_empty_page 以线性地址为参数，申请新的物理页面并完成映射过程。

/mm/memory.c

```
274 void get_empty_page(unsigned long address)
275 {
276 unsigned long tmp;
277
278    if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
279        free_page(tmp);                    /* 0 is ok - ignored */
280        oom();
281     }
282 }
283
```

其中 oom() 是用于内存使用完后的处理，显示完信息之后使调用进程退出。

/mm/memory.c

```
33 static inline volatile void oom(void)
34 {
35 printk( "out of memory\n\r");
36 do_exit(SIGSEGV);                    // 进程退出，出错码：SIGSEGV（资源暂时不可用）
37 }
38
```

5 释放页表：

内核使用了内存，自然就会有释放的时候。当进程创建时，需要获得大量的内存，也会释放大量的内存空间；当进程退出时，肯定有大量的内存需要释放。而伴随这种大量的内存释放工作，这些空间对应的页表也会变成无用的。如果不进行回收，将是巨大的浪费。

内核如果要做这种释放（见算法 free_page_tables），至少需要释放一个页表所映射的 4M 的线性空间，所以释放空间起始地址应该是以 4M 为边界的。要释放的空间不可以是低 16M 的空间。

```
算法：free_page_tables
输入：要释放空间起始线性地址 from
      要释放空间大小 size
输出：如果成功，返回 0；如果失败，使调用对象进入死循环
{
    if( 要释放的空间不是以 4M 为边界 )
        显示出错信息，调用对象死循环；
    if( 要释放的空间是用于内核控制物理内存的低 16M 空间 )
        显示出错信息，调用对象死循环；
    计算要释放的空间所占的页表数；
    for( 每个要释放的页表 ){
        for( 每个页表项 )
            if( 页表项映射有物理页面 )
                释放物理页面 free_page()；
            将该页表项设为空闲；
        }
        释放页表使用的物理页；
        将该页表对应的页目录项设为空闲；
    }
    刷新页变换高速缓冲；
    return 0；
}
```

因为这个线性空间是用于内核对物理内存的控制，不可以被释放。接下来要做的就很明显了。整个操作将导致页目录表的变化。由于 cpu 为了提高内存访问速度，会将页目录表和部分页表加载到 cpu 页变换高速缓存中，我们修改了页目录表就必须使 cpu 页变换高速缓存中的内容同我们修改后的相同，所以必须刷新页变换高速缓冲。通过重新对页目录表寄存器 cr3 赋值就可以使 cpu 刷新页变换高速缓冲。具体代码见下：

/mm/memory.c

```
39 #define invalidate() \
40     __asm__( "movl %%eax,%%cr3":: "a" (0))          // 寄存器 eax 中存放 0，即页目录表起始地址
41
101 /*
102 * This function frees a continuos block of page tables, as needed
103 * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
104 */
/*
* 下面函数释放页表连续的内存块，'exit()' 需要该函数。与 copy_page_tables()
* 类似，该函数仅处理 4Mb 的内存块。
```

```
*/
105 int free_page_tables(unsigned long from,unsigned long size)
106 {
107 unsigned long *pg_table;
108 unsigned long * dir, nr;
109
110  if (from & 0x3fffff)                                    // 要释放空间线性地址应该以 4M 为边界。
111            panic( "free_page_tables called with wrong alignment");
112  if (!from)                                              // 这里只对低 4M 空间的释放进行限制，BUG
113           panic( "Trying to free up swapper memory space");
114   size = (size + 0x3fffff) >> 22;                        // 计算要释放的页表数
115   dir = (unsigned long *) ((from>>20) & 0xffc);     /* _pg_dir = 0 */ // 第一个要释放页表对应的页目录项
116   for ( ; size-->0 ; dir++) {
117         if (!(1 & *dir))                                 // 该目录项是否有效
118               continue;
119        pg_table = (unsigned long *) (0xfffff000 & *dir);                  // 计算页表起始地址。
120        for (nr=0 ; nr<1024 ; nr++) {
121              if (1 & *pg_table)                         // 页表项有效，则释放对应页。
122                    free_page(0xfffff000 & *pg_table);
123             *pg_table = 0;                              // 将对应页表项置为空闲
124              pg_table++;
125         }
126          free_page(0xfffff000 & *dir);                  //  释放页表使用的物理页；
127          *dir = 0;                                      // 将对应页目录表项置为空闲
128      }
129  invalidate();                                          // 刷新页变换高速缓冲。
130  return 0;
131 }
132
```

6 内存共享

在一个系统中，内存往往是最紧张的资源，为了将这种资源合理的利用，就必须搞清楚内存是如何被使用的，只有这样才能进行合理而且高效的分配。

进程有时需要读内存，有时需要写内存，这个过程似乎是随机的。但是对于每一个进程来说，有一个段是不会被写的，那就是代码段。这个段的内容由进程的可执行文件决定，不会改变，进程运行时，也不能修改这个段的内容。如果两个进程使用的是同一个可执行文件，比如：打开两个 try.exe 文件（假定该可执行文件不支持多线程），这时如果让两个进程在内存中分别使用两份代码，将造成不必要的内存浪费，所以这种情况下，内核会让这两个进程使用同一个代码段的内存空间。

其实数据段也是可以共享的，同样是使用同一可执行文件的多个进程，他们在进程还没有开始执行时，数据段是相同的，随着进程的运行，数据段的内容可能会被改变。内存的访问具有局部性，在一段时间内可能不会有对数据段的某些区间修改，这个时间段可能很短，如果进程的数据操作完全靠堆栈来实现，这个时间段就可能是进程的整个生命周期。但是如何预测进程的数据操作是在哪里，如何预测数据段哪些区间可以共享，哪些不行，从而安排内存的使用？答案是否定的，对于现代操作系统而言，这种预测是不现实的，或者代价相当大。与其花费大量精力去做预测，为什么不采用以逸待劳的办法呢？先将空间共享，等到进程对共享空间进行写操作时再取消对该页的共享。liunx 采用了一种称为写时复制 (copy on write) 的机制。这种机制必须要有硬件的支持，在 386 页面映射机制中，有一个读写权限标志位 XW，将这一位设为只读（0）方式之后，如果进程试图对该页进行写操作，cpu 将出现页面异常中断，调用内核设定的页面异常中断处理程序，在这里内核将原来的页面复制一份，再取消对该页面的共享，这样就互不干扰了。有了这个保障，内核在进行内存共享操作时就可以放心了。

页表基地址的高 20 位		D	A	PCD	PWT	U	XW	P

当内核使用 fork 创建一个进程时，子进程将使用和父进程的同样代码段和数据段，然后根据 fork 返回的不同的值作为判断，运行不同的代码。见示例程序：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void main(){
      int childpid.data=100;
      childpid=fork();
      if(childpid==0){
          printf("I'm child!/n");
          printf("My father have a data ,it's %d!/n",data);
          exit(0);
        }
      printf("I'm father!I have a child %d/n",childpid);
      exit(0);
}
```

创建一个进程后，父进程和子进程使用同样的代码。但是他们中 childpid 的值不同，如果是子进程，childpid 的值是 0；如果是父进程，childpid 的值是子进程的进程号。在这以后，子进程可能会使用父进程中的一些数据。如果子进程不调用另一个可执行文件作为其执行代码，子进程将一直使用父进程的代码。

6.1 共享空间

有了 386 对页面共享的支持，共享空间的方法就很容易想到了。将被共享的空间的页目录表和页表复制一份，并且将所有页表项的访问属性设为只读，并修改页面映射表中的页面引用信息即可。具体算法如下：

```
算法：copy_page_tables
输入：共享源页面起始地址 from
      共享目的空间页面起始地址 to
      被共享空间的大小 size
输出：如果成功，返回 0
{
    if(from 或者 to 不是以 4M 为边界 )
      显示出错信息，使调用对象进入死循环；
    for( 共享源空间的每一个页目录项 ){
      if( 对应共享目的空间的页表已经存在 )
        显示出错信息，死循环；
      if( 共享源空间的页目录项不存在 )
        continue；
      为对应共享目的空间分配空闲页作为页表；
      设置该空闲页属性（可写、用户、有效）
      if( 共享源空间本次复制的是前 4M 的内核空间 )
        本次共享空间只是前 640K；
      for( 每个要共享空间的页表项 ){
        复制页表项；
        if( 对应页不存在 )
          continue；
        if( 被共享页在主内存块映射表映射范围内 ){
          将两个页表项都置为只读；
          对应页面映射项内容加 1；
        }
        else
          只将复制的页表项置为只读；
      }
    }
  刷新页变换高速缓冲；
}
```

对于带有页表复制，和带有页表的释放一样，必须保证被共享的空间和被共享到的空间起始地址是以 4M 为边界的；每次共享 4M 的空间，像以前一样，对于内核空间必须作特殊处理。640K 到 1M 的空间本来是高速缓冲块的空间，但是被显存和 BIOS 占用了，所以这部分空间是不共享的；因为 linus 当初使用的计算机有 16M 的内存，高速缓冲空间结束位置是 4M（见启动后内存分配），所以可能是由于这个原因，1M 到 3,071K 这个空间也是不共享的，对高速缓冲共享是没有意义的，这样内核的前 4M 空间就只共享 640K。如果被共享页不在主内存块映射表范围内，共享的就是这 640K 的空间，是内核使用的，在共享时，源页表项不被置为只读。

/mm/memory.c

132

133 /*

134 * Well, here is one of the most complicated functions in mm. It

```
135 * copies a range of linerar 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 */
/*
* 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！我们并不是仅复制任何内存块 - 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大小），因为这样处理可使函数很简单。不管怎样，
* 它仅被 fork() 使用（fork.c 第 56 行）。
*
* 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
* 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的浪费 - 我们
* 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（nr 在程序中指页面数）。
*/
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  if ((from&0x3fffff) || (to&0x3fffff))                    // 判断是否以 4M 为边界
159          panic( "copy_page_tables called with wrong alignment");
```

```
160   from_dir = (unsigned long *) ((from>>20) & 0xffc);     /* _pg_dir = 0 */     // 计算起始页目录项
161   to_dir = (unsigned long *) ((to>>20) & 0xffc);
162   size = ((unsigned) (size+0x3fffff)) >> 22;                          // 计算要共享的页表数
163 for( ; size-->0 ; from_dir++,to_dir++) {
164      if (1 & *to_dir)                                  // 被共享到的页表已经存在
165          panic( "copy_page_tables: already exist");
166      if (!(1 & *from_dir))                             // 被共享的页表不存在
167          continue;
168     from_page_table = (unsigned long *) (0xfffff000 & *from_dir);     // 取源页表地址
169     if (!(to_page_table = (unsigned long *) get_free_page()))
170             return -1;                                          /* Out of memory, see freeing */
171     *to_dir = ((unsigned long) to_page_table) | 7;            // 设置该页属性（可写、用户、有效）
172      nr = (from==0)?0xA0:1024;                                 // 如果是前 4M 空间，只共享 640K(160 页 )
173     for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
174          this_page = *from_page_table;
175          if (!(1 & this_page))                                // 如果当前源页表项没有使用，则不用复制
176           continue;
177          this_page &= ~2;                                      // 将目的页表项设为只读
178          *to_page_table = this_page;
179          if (this_page > LOW_MEM) {                          // 如果被共享页在主内存块映射表映射范围内
180              *from_page_table = this_page;                             // 源页表项设为只读
181              this_page -= LOW_MEM;
182              this_page >>= 12;
183              mem_map[this_page]++;                                  // 共享数加一
184              }
185         }
186  }
187 invalidate();                                                       // 刷新页变换高速缓冲。
188 return 0;
189 }
190
```

6.2 共享进程空间

6.2.1 早期共享

当内核使用 fork 创建一个进程时，子进程将使用和父进程的进程空间进行完全的拷贝。子进程除了要与父进程共享内存空间外，如果要在这个内存空间上运行还需要根据父进程的数据段描述符和代码段描述符设置子进程的自己的数据段描述符和代码段描述符。算法如下：

```
算法：copy_mem
输入：子进程进程号 nr
      子进程进程控制块 p
输出：如果成功，返回 0
{
      取得父进程的数据段、代码段的段限长和基地址；
      if（数据段和代码段段限长和基地址不合法）
          显示出错信息，死循环；
      设置子进程的数据段、代码段的段限长和基地址；
      共享代码段和数据段内存空间（copy_page_tables）
      if（共享失败）{
          释放子进程共享内存空间时申请的页面；
          return 共享失败；
       }
       return 0；
}
```

由于 linux0.11 只支持数据段和代码段基址相同的进程，所以判断数据段和代码段的合法性首先应该检测两者是否相同；又由于代码段在数据段之前，所以代码段限长一定要小于数据段限长；

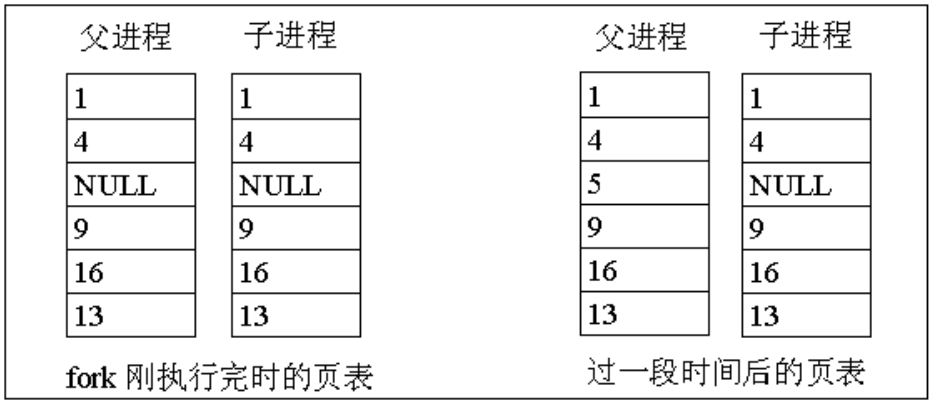
/kernel/fork.c

```
    // 当前进程即父进程
39 int copy_mem(int nr,struct task_struct * p)
40 {
41 unsigned long old_data_base,new_data_base,data_limit;
42 unsigned long old_code_base,new_code_base,code_limit;
43
44  code_limit=get_limit(0x0f);                    // 取当前进程代码段和数据段段限长
45  data_limit=get_limit(0x17);
46  old_code_base = get_base(current->ldt[1]);      // 取原代码段和数据段段基址
47  old_data_base = get_base(current->ldt[2]);
48  if (old_data_base != old_code_base)
49          panic( "We don't support separate I&D");
50  if (data_limit < code_limit)
51          panic( "Bad data_limit");
52  new_data_base = new_code_base = nr * 0x4000000;     // 子进程基址 = 进程号 *64Mb( 进程线性空间 )
53  p->start_code = new_code_base;
54  set_base(p->ldt[1],new_code_base);                  // 设置代码段、数据段基址
55  set_base(p->ldt[2],new_data_base);
56  if (copy_page_tables(old_data_base,new_data_base,data_limit)) {      // 共享代码段和数据段内存空间
```

```
57          free_page_tables(new_data_base,data_limit);          // 释放共享内存空间时申请的页面
58          return -ENOMEM;
59      }
60  return 0;
61 }
62
```

6.2.2 后期共享

当子进程被 fork 出来后，就会和父进程分道扬镳，独立地被内核调度执行，在这个过程中父进程和子进程的执行是独立的，互不影响。如果父进程因为缺页新申请了物理页面，子进程是不知道的。示例如下：



当子进程产生缺页时，子进程还是要尽量地“偷懒”，除了在被 fork 出来时可以与父进程共享内存外，父进程新申请的物理页也是可以被共享的。只要申请页被读入之后还没有被改变过就可以共享。

其实上面说的例子中，如果是子进程申请了新的物理页，父进程同样可以拿来用，如果子进程还 fork 了孙进程，孙进程申请的页面子进程和父进程都可以使用。因为分道扬镳之后各个进程是平等的，只要大家都使用同一个可执行程序，谁先申请新物理页都是一样的。

试图共享内存的算法如下：

```
算法：share_page
输入：共享地址 address
输出：如果成功，返回 1
{
    if ( 要求共享的进程 A 没有对应的可执行文件 )
        return 0；
    if (A 对应的可执行文件没有被多个进程使用 )
        return 0；
    for( 每个存在的进程 P)
      {
        if (P 就是要求共享的进程本身 )
            continue；
        if (P 对应可执行文件与要求共享的进程的不同 )
            continue；
        if (P 进程共享地址对应的物理页不存在或不干净 )
            continue；
        if ( 对应物理页不属于主内存块 )
            continue；
        if ( 进程 A 共享地址对应的页表不存在 )
```

```
            {
            为页表分配新的物理页；
            设置页目录项；
             }
          if ( 进程 A 对应的页表项已经存在 )
              显示错误信息，死循环；
          将进程 P 对应的页表项属性设为只读；
          设置进程 A 对应地址的页表项；
          物理页引用数加 1；
          刷新页变换高速缓冲。
           return 1；
         }
       return 0；
}
```

对于每一个进程都应该对应一个可执行文件，当进程处于某些特定时刻（如：正在作进行初始化设置）时没有对应的可执行文件，当然也就不应该作共享处理。如果对应的可执行文件应用数不大于 1，则表示没有进程与要求共享的进程共享对应的可执行文件，也不会有共享对象。

接下来的任务就是找到一个符合要求的共享物理页，条件有：1，进程对应可执行文件相同；2，对应物理页在被读入之后没有被修改过。如果要求共享进程对应地址的页表项存在，但是原来是因为缺页才进入共享操作的，肯定系统出现了严重错误。

最后进程 P 对应的页表项属性修改为只读，设置进程 A 对应地址的页表项，使它指向共享物理页，属性为只读，物理页对应主内存块映射数组项加 1；因为页表发生了变化，所以要刷新页变换高速缓冲。下面是具体代码：

/mm/memory.c

```
335
336 /*
337 * share_page() tries to find a process that could share a page with
338 * the current one. Address is the address of the wanted page relative
339 * to the current data space.
340 *
341 * We first check if it is at all feasible by checking executable->i_count.
342 * It should be >1 if there are other tasks sharing this inode.
343 */
/*
* share_page() 试图找到一个进程，它可以与当前进程共享页面。参数 address 是
* 当前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。如果有其它任务已共享
* 该 inode，则它应该大于 1。
*/
344 static int share_page(unsigned long address)
345 {
```

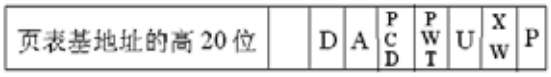
```
346 struct task_struct ** p;
347
348  if (!current->executable)                    // 没有对应的可执行文件
349          return 0;
350  if (current->executable->i_count < 2)              // 不是多进程共享可执行文件
351     return 0;
352   for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {                  // 搜索每个进程控制块指针
353           if (!*p)                                    // 没有对应进程
354             continue;
355           if (current == *p)                          // 就是指向当前任务
356              continue;
357           if ((*p)->executable != current->executable)          // 不是与当前任务使用同一个可执行文件
358              continue;
359           if (try_to_share(address,*p))                       // 试图共享页面
360              return 1;
361      }
362  return 0;
363 }
364
```

下面是 try_to_share(address,*p) 的代码：

/mm/memory.c

```
283
284 /*
285 * try_to_share() checks the page at address "address" in the task "p",
286 * to see if it exists, and if it is clean. If so, share it with the current
287 * task.
288 *
289 * NOTE! This assumes we have checked that p != current, and that they
290 * share the same executable.
291 */
/*
* try_to_share() 在任务 "p" 中检查位于地址 "address" 处的页面，看页面是否存在，是否干净。
* 如果是干净的话，就与当前任务共享。
*current 共享 p 已有的物理页面
* 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。
*/
// address 是线性地址，是一个相对于 code_start 的偏移量，在执行完这个函数后，p 和 current 偏移是
// address 的位置共享物理内存 !!!
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
300 from_page = to_page = ((address>>20) & 0xffc);          // 计算相对于起始代码偏移的页目录项数
 // 加上自身的 start_code 的页目录项，得到 address 分别在 p 和 current 中对应的页目录项
301 from_page += ((p->start_code>>20) & 0xffc);
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？*/
304 from = *(unsigned long *) from_page;                    // 取页目录项的内容
305   if (!(from & 1))                                      // 对应页表是否存在
306       return 0;
 // 取对应的页表项
307  from &= 0xfffff000;
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？*/
311  if ((phys_addr & 0x41) != 0x01)
312       return 0;
313  phys_addr &= 0xfffff000;
314  if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)      // 是否在主内存块中
315       return 0;
```




```
// 取页目录项内容 ..to。如果该目录项无效 (P=0)，则取空闲页面，并更新 to_page 所指的目录项。
316  to = *(unsigned long *) to_page;                     // 取目标地址的页目录项
317  if (!(to & 1))                                       // 如果对应页表不存在
318       if (to = get_free_page())                       // 分配新的物理页
319           *(unsigned long *) to_page = to | 7;
320       else
321           oom();
322   to &= 0xfffff000;                                   // 取目标地址的页表项
323   to_page = to + ((address>>10) & 0xffc);
324   if (1 & *(unsigned long *) to_page)                 // 如果对应页表项已经存在，则出错，死循环
325       panic( "try_to_share: to_page already exists");
326   /* share them: write-protect */
      /* 对它们进行共享处理：写保护 . */
```

```
327  *(unsigned long *) from_page &= ~2;
328  *(unsigned long *) to_page = *(unsigned long *) from_page;          共享物理内存
//    刷新页变换高速缓冲。
329 invalidate();
// 共享物理页引用数加 1
330 phys_addr -= LOW_MEM;
331 phys_addr >>= 12;
332 mem_map[phys_addr]++;
333 return 1;
334 }
335
```

7 页面异常

当 cpu 在进行内存访问时，可能因为缺页或者试图对一个只读页面进行写操作而产生页面异常，cpu 进入相应的页面异常中断处理程序。

由于异常可能由缺页或者写只读页面产生，两种情况的处理也是不同的，所以中断处理程序首先应该区分产生本次异常的原因，进入不同的处理过程。算法如下：

```
算法：page_fault
输入：出错码 error_code
      出错线性地址 address
输出：无
{
    保存现场；
    根据出错码判断出错原因；
    if ( 缺页 )
        作缺页处理do_no_page(error_code, address);
    else
        作写保护出错处理 do_wp_page(error_code, address)；
    恢复现场；
    return；
}
```

在 x86 处理器中 error_code 由 cpu 产生并在保存了中断点的相关内容之后将其压入堆栈，出错码的最低位指示出错原因（1：写出错；0：缺页）。address 则是由一个专门的 32 位寄存器 cr2 保存。具体代码如下：

/mm/page.s

```
11
12 .globl _page_fault
13
14 _page_fault:
15      xchgl %eax,(%esp)          // 交换 eax 与 esp 所指向空间的内容 =>1. 保存 eax; 2. 取出 error_code
16      pushl %ecx                 // 保存现场
17      pushl %edx
18      push %ds
19      push %es
20      push %fs
21      movl $0x10,%edx             //(21 ～ 24 行 ) 使 ds、es、fs 指向系统数据段
22      mov %dx,%ds
23      mov %dx,%es
24      mov %dx,%fs
25      movl %cr2,%edx              // 取出错线性地址
26      pushl %edx                  // 将出错地址和出错码压入堆栈，作为处理函数的输入参数
27      pushl %eax
28      testl $1,%eax              // 判断出错码最低位，决定调用函数
```

```
29      jne 1f                          // 为 1，调用写保护出错处理函数
30      call _do_no_page                // 为 0，调用缺页处理函数
31      jmp 2f
32  1: call _do_wp_page
33  2: addl $8,%esp                     // 丢弃输入参数 error_code 和 address
34      pop %fs                         // 恢复现场
35      pop %es
36      pop %ds
37      popl %edx
38      popl %ecx
39      popl %eax
40      iret
```

在这段代码中，我们可以充分领略到系统程序员对汇编编程知识的要求。 在第 15 行 xchgl %eax,(%esp) ，必须非常清楚压栈过程。当 cpu 执行压栈操作时，是先执行 esp=esp-4; 再将数据送入 esp 所指向的单元。cpu 在进入异常中断处理程序之前，将 error_code 压入了堆栈，当前 esp 指向的单元存放的就是 error_code，所以第 15 行的命令，取出了 error_code 又将 eax 保存了，如果要用其他方法实现应该是

```
pushl eax
movl (esp+4),eax
```

相比之下，15 行的程序将 eax 放在了 error_code 原来存放的空间，节约了堆栈空间，同时也节约指令数，可谓是一箭四雕。在 33 行 2: addl $8,%esp 对于输入参数的丢弃，不是用两次 popl 操作，而是直接将 esp 加 8，又省了一条指令。可见高水平的系统程序员为了提高效率是多么的抠门。这样的程序虽然效率高，但是对于理解会有一定的障碍，不过换个方向来想，毕竟这种底层代码不是人人都会去仔细读的。

7.1 缺页中断

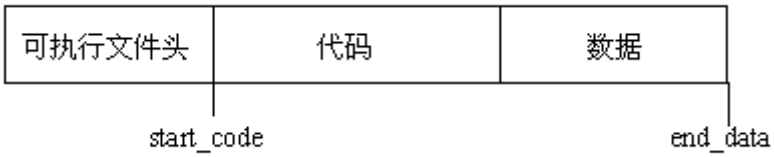
在对进行进程初始设置时，内核并不是将进程可能用到的所有内存一次性分配给进程，而是在进程要访问该地址时分配，将内存分配给一定会被访问的空间，这样就提高内存资源的使用率。这样作就不可避免地会出现缺页中断。当 cpu 访问一个内存单元时，如果该单元所在的页面不在内存中，cpu 将产生页面异常，进一步进入缺页处理程序，算法如下：

```
算法：do_no_page
输入：出错码 error_code
      出错线性地址 address
输出：无
{
    if ( 出错进程没有对应的可执行文件
        || 出错地址不在代码和数据段 )
        {
          分配物理页面并映射到出错线性地址 ( 使用 get_empty_page())；
          return；
        }
    试图共享页面 ( 使用 share_page())；
    if ( 共享页面成功 )
        return ；
```

```
        分配新的物理页面 (get_free_page())；
        从可执行文件中将页面对应的内容读入内存；
        将页面中不属于代码段和数据段的内容清零；
        将新的物理页面映射到出错线性地址 (put_page())；
        if ( 映射失败 )
         {
          释放新申请的物理页面；
          显示出错，死循环；
          }
        return；
}
```

进程在不同的时刻会处于不同的状态，如果进程此时还处于初始化时期，就可能还没有设置对应的可执行文件，这个时候的内存使用请求可能是与其设置有关的，所以需要为其分配内存。

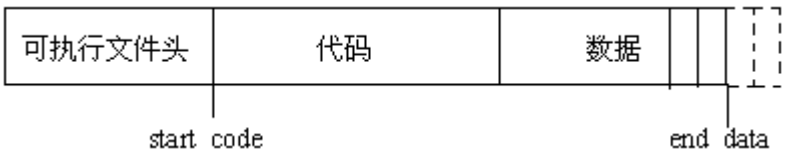
对于进程的可执行文件，在这里只是说一下它的基本结构：



进程对应的可执行文件包含有进程的代码段和数据段的内容，进程的线性地址与可执行文件内容的逻辑地址是对应的，如果出错是在代码段和数据段，应该先试图共享内存，共享不成功就应该分配内存并从可执行文件中读取相应内容，；如果不是在这两个段，就直接分配内存。

可执行文件存储在磁盘上，磁盘的存储基本单位是 1KB，所以要读取一个页面的内容就要读取四个磁盘块。从可执行文件中读取内容由 bmap 和 bread_page 两个函数来作，首先将出错线性地址所在页面换算成可执行文件对应的逻辑盘块号，bmap 用于将逻辑盘块号换算成物理盘块号，最后由 bread_page 将四个物理盘块读入内存。

在读入过程中，可能出现这种情况，由于线性地址太大，对应页面换算得到的逻辑盘块号过大，对应可执行文件却没有这么大（如下所示，这时后两块逻辑盘块将不会被读入），多余的逻辑盘将不会被读入。



另外，读入一页内存之后，该页的结束地址可能会超过 end_data。

由于上述两种情况，应该对多出来的内存清零。

最后就是映射页面，put_page 只有在申请新的页表空间失败的情况会返回 0，这时就应该将已经申请了的物理页面释放，然后调用 oom() 报错。

具体代码如下：

```
  /mm/memory.c
365 void do_no_page(unsigned long error_code,unsigned long address)
366 {
367     int nr[4];
368     unsigned long tmp;
```

```
369     unsigned long page;
370      int block,i;
371
372     address &= 0xfffff000;                      // 取 页面起始地址
373     tmp = address - current->start_code;        // 换算出相对于进程代码起始地址的相对地址
374     if (!current->executable || tmp >= current->end_data) {    // 没有对应可执行文件或不在代码和数据段
375         get_empty_page(address);
376         return;
377     }
                // 到此处时一定是代码和数据长度范围内
378     if (share_page(tmp))                                       // 尝试共享内存
379         return;
380     if (!(page = get_free_page()))                             // 分配新的物理内存
381         oom();
382 /* remember that 1 block is used for header */
    /* 记住，（程序）头要使用 1 个数据块 */
383     block = 1 + tmp/BLOCK_SIZE;                                // 换算逻辑块起始块号
384     for (i=0 ; i<4 ; block++,i++)                              // 将逻辑块号换算成物理块号
385         nr[i] = bmap(current->executable,block);
386     bread_page(page,current->executable->i_dev,nr);            // 读入一个页的四个磁盘块
387     i = tmp + 4096 - current->end_data;                        // 对超出数据段的内容清零
388     tmp = page + 4096;
389     while (i-- > 0) {
390         tmp--;
391         *(char *)tmp = 0;
392     }
393     if (put_page(page,address))                                // 页面映射
394         return;
395   free_page(page);
396   oom();
397 }
398
```

7.2 页面写保护错误

由于进程的 fork 和 share_page 操作，会出现多个进程共享一个物理页面的情况，这个物理页面被置为只读方式，如果其中一个进程想对这个页面进行写操作，cpu 就会产生页面异常中断，并进一步进入写保护出错处理。

在写保护出错处理中，将会根据情况复制被共享的页或者取消对页面的写保护。

```
算法：do_wp_page
输入：出错码 error_code
      出错线性地址 address
输出：无
{
    if ( 出错地址属于进程的代码段 )
        将进程终止；
    if ( 出错页面属于主内存块且共享计数为 1)
     {
        取消写保护；
        刷新页变换高速缓冲；
        return；
     }
    申请一个新的物理页；
    if ( 出错页面属于主内存块 )
        共享计数减 1；
    使出错时的页表项指向新的物理页；
    刷新页变换高速缓冲；
    复制共享页的内容到新的物理页；
    return；
}
```

对于一般情况，对代码段的写操作是违法的，肯定是进程本身代码有问题，为了不进一步引起错误，系统将会把该进程终止。但是 estdio 库（后来不再为 linux 所使用）支持对代码段的写操作，linus 当时由于没有得到这个库的具体资料，所以也只是预留了操作。

一个页面被多个进程共享，每当一个进程产生一次写保护 错误，内核将给进程分配一个新的物理页面，将共享页面的内容复制过来，新的页面将设置为可读写，而共享页面仍然是只读的，只是共享计数减小了。当其他共享进程都产生了一次写保护错误后，共享页面的共享计数减成了 1，其实就是被一个进程独占了，但此时该共享页面仍然是只读的，如果独占它的进程对它进行写操作仍然会产生写保护出错。为什么不在共享计数减成了 1 之后就将共享页面置为可写呢？原因很简单，因为系统并不知道最后是哪个页表项指向这个共享页，如果要把它查找出来会有很大的系统开销，这是中断处理程序应当尽量避免的，所以采用了以逸待劳的办法。

如果当初共享的页面不属于主内存块，在共享时就没有作共享计数的处理，就不存在共享计数的问题，直接复制就可以了。

/mm/memory.c

239

240 /*

241 * This routine handles present pages, when users try to write

242 * to a shared page. It is done by copying the page to a new address

243 * and decrementing the shared-page counter for the old page.

244 *

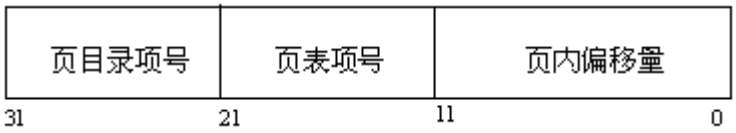
245 * If it's in code space we exit with a segment error.

246 */

/*

* 当用户试图往一个共享页面上写时，该函数处理已存在的内存页面，（写时复制）

```
* 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。
*
* 如果它在代码空间，我们就以段错误信息退出。
*/
247 void do_wp_page(unsigned long error_code,unsigned long address)
248 {
249   #if 0
250    /* we cannot do this yet: the estdio library writes to code space */
251    /* stupid, stupid. I really want the libc.a from GNU */
       /* 我们现在还不能这样做：因为 estdio 库会在代码空间执行写操作 */
       /* 真是太愚蠢了。我真想从 GNU 得到 libc.a 库。*/
252    if (CODE_SPACE(address))                  // 出错地址属于进程的代码段，则终止当前程序
253        do_exit(SIGSEGV);
254  #endif
// 处理取消页面保护。
// 输入参数指向出错页的页表项的指针
// 计算方法：
//     页表偏移量 + 页表起始地址
255  un_wp_page((unsigned long *)
256 (((address>>10) & 0xffc) + (0xfffff000 &*((unsigned long *) ((address>>20) &0xffc)))));
257
258
259  }
260
```



un_wp_page 的具体代码也在 /mm/memory.c 中

```
220
221 void un_wp_page(unsigned long * table_entry)
222 {
223    unsigned long old_page,new_page;
224
225    old_page = 0xfffff000 & *table_entry;          // 取出错页面对应的物理地址
                                                     // 如果属于主内存块且共享计数为 1
226   if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
227         *table_entry |= 2;                         // 共享页置为可写
228         invalidate();
229         return;
230   }
231   if (!(new_page=get_free_page()))                 // 申请一个空闲物理页面
232       oom();
```

```
233    if (old_page >= LOW_MEM)                    // 如果在主内存块中，将共享数减 1
234        mem_map[MAP_NR(old_page)]--;
235    *table_entry = new_page | 7;            // 改变 table_entry 的指向从而实现共享的分离
236    invalidate();
237    copy_page(old_page,new_page);        // 拷贝共享页
238  }
239
```

8 桶结构

8.1 桶结构定义与初始化

在 C 语言中，malloc（int size）用于在用户空间为某个大小为 size 的结构申请一块相应的内存空间，在内核中很多时候也需要为内核的数据结构分配内存空间。对于内核来说，用户进程分配内存空间的操作与用户的普通的内存访问操作并没有区别，在用户分配空间时，内核只需要关心是否超出了用户的虚拟地址空间就行了。内核给自己分配内存空间就不同了，内核要对物理内存进行管理，对这种小量的内存分配尤其需要仔细管理。

要管理，干脆就将将一部分内存页分成指定大小的块，大小是 1、2、3、4、5 …，当内核申请时就从含有对应大小块的内存中取出一块分配出去，比如：当内核申请一个大小为 3 的空间时，就从被分成了大小为 3 的块的内存页中取一个块来分配。这就是桶结构结构的基本思想。将内存页看成是桶，桶里面装的是相同大小的内存块。

如果使用 1、2、3、4 …这样的内存块大小进行管理，页面的利用率会很低，可能一个页面只会被申请很少几个结构，这样将造成很大的内存浪费。块的分配很大程度决定了桶结构的空间使用效率，经过理论计算和长期的实际运行证明，采用 2 的幂次作为块的大小具有理想的效果。linux0.11 采用的就是这种分块方式，根据实际情况，从 16、32、64 …一直到一个页的大小 4096 B，在分配时分配能满足需求的最小的块。

对于每个桶，应该有一个结构记录它的信息，以及与其他桶之间的关系。linux0.11 中定义了一个叫桶描述符的结构，具体结构如下：

/lib/malloc.c

```
52 struct bucket_desc {                  /* 16 bytes */     /* 本结构占 16 个字节 */
53     void *page;                      // 对应页面起始地址
54     struct bucket_desc *next;        // 指向下一个描述符的指针
55     void *freeptr;                   // 指向本桶中空闲内存块链表的指针
56     unsigned short refcnt;           // 引用计数
57     unsigned short bucket_size;      // 本描述符对应内存块的大小
58 };
```

为了方便查找，内核分别将相同块大小的桶串成一条链，链表头结构如下：

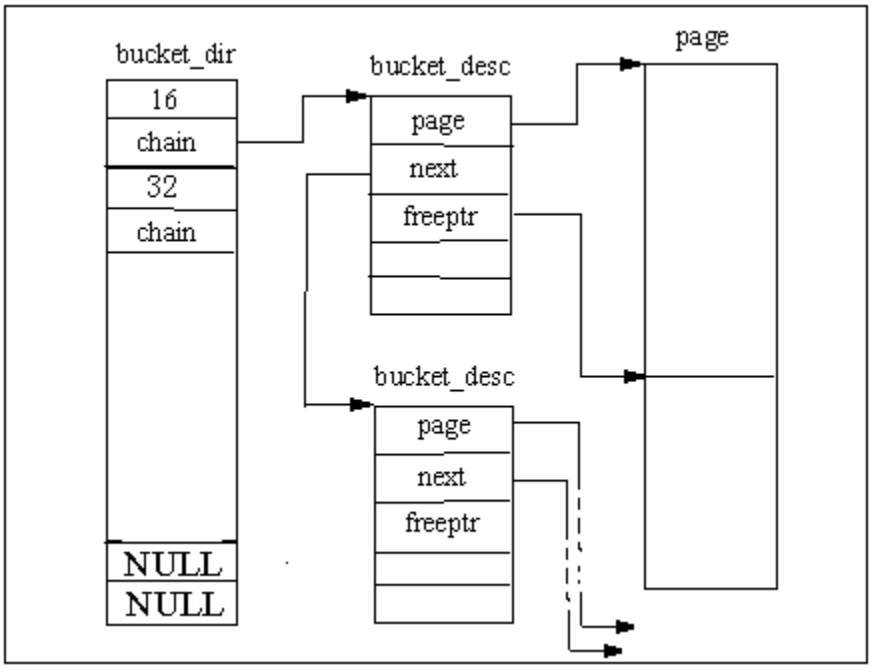
```
60 struct _bucket_dir {                  /* 8 bytes */   /* 本结构占 8 个字节 */
61         int size;                    // 本链中桶对应的内存块的大小 ( 字节数 )
62         struct bucket_desc *chain;   // 本链中的桶描述符链表头指针
63 };
```

同时还有一个数组用于保存所有的链表头：

```
77 struct _bucket_dir bucket_dir[] = {
78         { 16, (struct bucket_desc *) 0},        // 16 字节长度的内存块。
79         { 32, (struct bucket_desc *) 0},        // 32 字节长度的内存块。
80         { 64, (struct bucket_desc *) 0},        // 64 字节长度的内存块。
81         { 128, (struct bucket_desc *) 0},       // 128 字节长度的内存块。
82         { 256, (struct bucket_desc *) 0},       // 256 字节长度的内存块。
83         { 512, (struct bucket_desc *) 0},       // 512 字节长度的内存块。
84         { 1024, (struct bucket_desc *) 0},      // 1024 字节长度的内存块。
```

```
85          { 2048, (struct bucket_desc *) 0},              // 2048 字节长度的内存块。
86          { 4096, (struct bucket_desc *) 0},              // 4096 字节 (1 页 ) 内存。
87          { 0, (struct bucket_desc *) 0}
          };                        /* End of list marker */
88
```

这几级结构关联的结果如图：

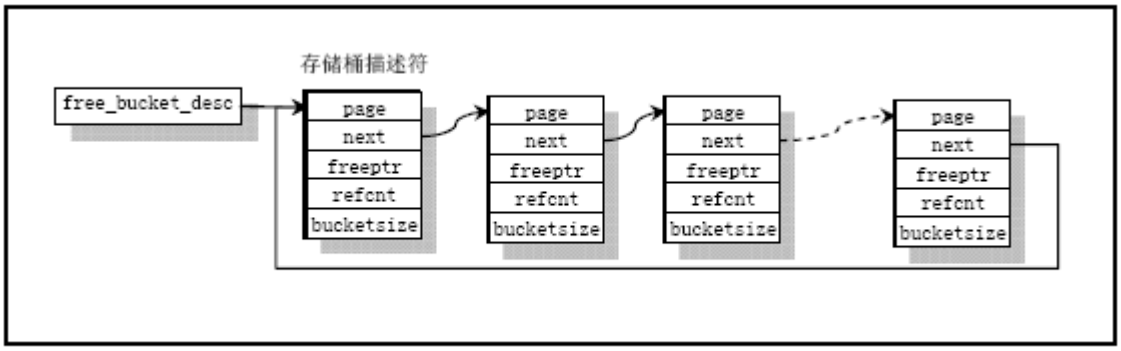


在初始化时，系统并不分配任何内存作为桶，而是在请求发生时才分配内存。在分配内存作为桶时，也要分配一个 bucket_desc 结构，如果这个结构的分配与一般的内核分配请求一样，就有可能进入一个死循环：分配函数自身也在请求分配。所以 对于 bucket_desc 结构需要独立的分配机制。

专门分配一个页用于存放 bucket_desc，当需要时就从这个页面中分配一个 bucket_desc 结构。在初始化时，将所有结构利用他们自身的 next 指针连成一个链表，用一个指针指向链表头。这个指针的定义为：

```
92 struct bucket_desc *free_bucket_desc = (struct bucket_desc *) 0;
```

这样就连成了下图的结构：



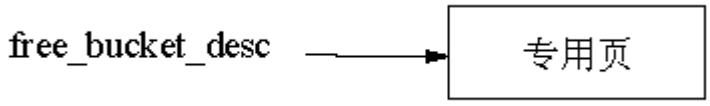
引自《linux0.11完全注释》

bucket_desc 专用页的初始化函数的算法为：

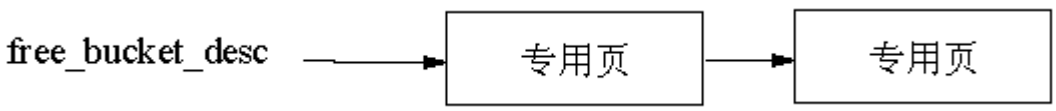
```
算法： init_bucket_desc
输入：无
输出：无
{
    申请一个空闲页（get_free_page）；
    for（每一个 bucket_desc 结构）
    {
        if（不是最后一个 bucket_desc 结构）
            next 指向下一个 bucket_desc 结构；
    }
    使最后一项的 next 指针指向 free_bucket_desc 指向的内容；
    使 free_bucket_desc 指向第一个 bucket_desc 结构；
    return ；
}
```

一般情况下，这个初始化程序是在链表为空，即 free_bucket_desc 为 NULL 时才会被调用。内核中的数据结构，尤其是链表，需要考虑多任务调度可能造成内核数据结构不一致的情况，对于 get_free_page 这种操作，如果系统引入了虚拟内存管理之后，很可能会导致当前执行进程进入睡眠等待内存，在这个进程被激活之前，在其他进程执行时，内核可能会因为结构分配而再次调用到 init_bucket_desc，如果这时顺利执行了，专用页被分配了，在睡眠的进程重新运行时，仍然会去改变 bucket_desc 链表。为了避免出错，在程序结构上要特别处理。一种方案是：在改变链表前检查 free_bucket_desc 是否还是 NULL，如果不是，就成功退出，如果是，就该进行对链表的操作；linux0.11 并没有采用这种方案，而是继续对链表操作，最后并不是直接的对 free_bucket_desc 赋值，而是向链表头部加入内容的操作，这样就避免内核数据结构的不一致。

我们假设进程 A 执行时系统进入 init_bucket_desc，并在申请页面（get_free_page）时被阻塞。之后进程 B 运行，系统又进入 init_bucket_desc，顺利执行完后退出，这时内核中空闲 bucket_desc 链表如下：



进程 A 恢复执行后，从 get_free_page 中返回，继续执行完 init_bucket_desc 函数，经过最后两个操作（向链表头部加入内容的操作）之后将两个页面连接到了一起：



当读到后面 8.2 的代码后，会发现这些考虑在 linux0.11 中并不需要，这正是体现了 linux 设计的严谨。

/lib/malloc.c

93

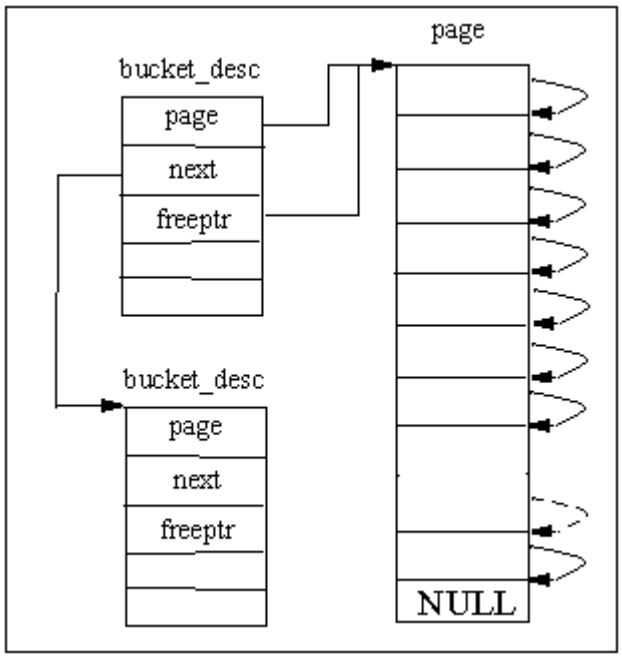
94 /*

95 * This routine initializes a bucket description page.

96 */

/*

* 下面的程序用于初始化一页桶描述符页面。
*/
```
97 static inline void init_bucket_desc()
98 {
99     struct bucket_desc *bdesc, *first;
100   int i;
101
102   first = bdesc = (struct bucket_desc *) get_free_page();          // 申请一页内存 ,first 指向页基址
103    if (!bdesc)
104         panic( "Out of memory in init_bucket_desc()");
105    for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {      // 将页中的 bucket_desc 结构连成链表
106             bdesc->next = bdesc+1;
107             bdesc++;
108     }
109 /*
110 * This is done last, to avoid race conditions in case
111 * get_free_page() sleeps and this routine gets called again....
112 */
```
/*
* 这是在最后处理的，目的是为了避免在 get_free_page() 睡眠时该子程序又被
* 调用而引起的竞争条件。
*/
```
 // 将新页的链表连入系统的空闲 bucket_desc 链表
113   bdesc->next = free_bucket_desc;
114   free_bucket_desc = first;
115 }
116
```
在桶页面内部，所有的空闲内存块也是串成了一个链表，桶描述符中的 freeptr 指向链表头。使用下面的代码初始化桶页面就有了右图所示的结构：




```
          (void *) cp = get_free_page();
160       for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
161           *((char **) cp) = cp + bdir->size;
162           cp += bdir->size;
163       }
164        *((char **) cp) = 0;
```

8.2 桶内存块的分配

在标准的 C 函数中使用 malloc（int len）进行内存块分配，在 linux0.11 中使用了一个同名的函数 malloc（int len）进行内核的内存块分配。为了避免混淆，在 0.98 版内核之后采用了 kmalloc（int size）。相对应的 linux0.11 中的内核内存块释放函数 free_s 也改名为 kfree_s。

了解了桶的组织结构，分配内存块的算法就不难理解了。

```
算法：malloc
输入：申请内存块大小 len
输出：如果成功，返回内存块指针；失败则返回 NULL；
{
    查找一个桶链表，链表中 桶的内存块是能够满足要求的最小块；
   if（没有搜索到符合要求的链）
      {
          打印出错信息：请求块过大；
          进入死循环；
       }
    关中断；
    在链表中查询还有空闲内存块的桶；
    if（链表中所有桶都没有空闲的内存块）
      {
          if（没有空闲桶描述符）
              初始化一个页面用作桶描述符（init_bucket_desc）；
          从空闲桶描述符链表中分配一个桶描述符；
          分配一个新物理页面作为桶；
          初始化桶页面；
          设置桶描述符指针；
          将新桶连入对应的链表；
      }
    从桶中分配一个空闲内存块；
    开中断；
     return  空闲内存块指针；
}
```

桶结构最大能够分配的内存块的大小是 4KB，如果这个大小都还不能满足要求，那么肯定是出问题了。内核中不可能申请这么大的内存块。

之后的操作是在中断关闭的状态下执行的，所以就不会因为进程切换导致数据结构不一致的情况。似乎 init_bucket_desc 中关于避免进程切换的安排是多余的，但是使用关中断作为避免竞争的手段是低效的，尤其是在中间执行过程复杂的情况下。如果以后引入虚存管理之后，可能必须有进程切换，到时就不能使用关中断了。所以提前在程序结构上作安排才是正确之道。

/lib/malloc.c

```
116
117 void *malloc(unsigned int len)
118 {
119    struct _bucket_dir *bdir;
120    struct bucket_desc *bdesc;
121    void *retval;
```

```
122
123 /*
124 * First we search the bucket_dir to find the right bucket change
125 * for this request.
126 */
  /*
   * 首先我们搜索存储桶目录 bucket_dir 来寻找适合请求的桶大小。
   */
127  for (bdir = bucket_dir; bdir->size; bdir++)              // 寻找合适的链表：(bdir->size)/2<len<=bdir->size
128        if (bdir->size >= len)
129              break;
130  if (!bdir->size) {                                       // 链表头数组最后一项是 NULL
131        printk( "malloc called with impossibly large argument (%d)\n",  len);
132
133        panic( "malloc: bad arg");
134  }
135 /*
136 * Now we search for a bucket descriptor which has free space
137 */
  /*
   * 现在我们来搜索具有空闲空间的桶描述符。
   */
138   cli();                        /* Avoid race conditions */ /* 为了避免出现竞争条件 */   // 关中断
139   for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)          // 寻找还有空闲内存块的桶
140         if (bdesc->freeptr)
141            break;
142 /*
143 * If we didn't find a bucket with free space, then we'll
144 * allocate a new one.
145 */
 /*
  * 如果没有找到具有空闲空间的桶描述符，那么我们就要新建立一个该目录项的描述符。
  */
146   if (!bdesc) {
147         char *cp;
148         int i;
149
150       if (!free_bucket_desc)                          // 没有空闲桶描述符
151            init_bucket_desc();
152       bdesc = free_bucket_desc;                       // 取一个空闲桶描述符
```

```
153       free_bucket_desc = bdesc->next;
154       bdesc->refcnt = 0;                          // 设置新的桶描述符
155       bdesc->bucket_size = bdir->size;
156       bdesc->page = bdesc->freeptr = (void *) cp = get_free_page();
157       if (!cp)                                    // 页面申请失败
158           panic( "Out of memory in kernel malloc()");
159 /* Set up the chain of free objects */
/* 在该页空闲内存中建立空闲对象链表 */
160       for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
161           *((char **) cp) = cp + bdir->size;
162           cp += bdir->size;
163       }
164       *((char **) cp) = 0;          最后一项的指针为 NULL
165       bdesc->next = bdir->chain;          /* OK, link it in! */ /* OK，将其链入！*/
166       bdir->chain = bdesc;
167   }
                                  // 分配一个空闲内存块
168   retval = (void *) bdesc->freeptr;
169   bdesc->freeptr = *((void **) retval);
170   bdesc->refcnt++;
171   sti();                    /* OK, we're safe again */ /* OK，现在我们又安全了 */       // 开中断
172   return(retval);
173 }
174
```

8.3 桶内存块的回收

当使用完内存块之后，内核将释放内存块。申请时内核知道需要的内存块的大小，在释放时却不一定知道要释放的这块内存的大小是多少。如果没有指定要释放的内存块的大小，怎么确定要释放内存块的大小呢？在释放时必然会指定内存块的地址，这个地址唯一的对应了一个物理页面，也就是唯一对应了一个桶，这个桶的描述符中就记录了内存块的大小。在 linux0.11 中使用 free_s(void *obj, int size) 来释放内存块，如果不指定大小，输入参数 size 是 0

```
算法：free_s
输入：释放对象指针obj
        释放对象大小 size（如果是 0，表示没有指定大小）
输出：无
{
      for（每一个桶链表）{
          if（链表中桶的内存块大小 < 释放对象大小）
              continue；
          for（每一个桶）
              if（是释放对象对应的桶）
                  退出搜索；
```

```
    }
    if（搜索桶失败）
        显示出错信息，死循环；
    关中断；
    将要释放的内存块链入桶的空闲链表；
    修改桶的相关信息；
    if（桶中所有的内存块都是空闲的）
    {
        释放桶对应的内存块；
        释放桶对应的描述符；
    }
    开中断；
    return；
}
```

可以看出在搜索时，如果输入参数 size 为 0，将会一次搜索不同 size 的桶链表。如果指定了 size，搜索速度就会大幅度提高。但在内核中仍然提供了对不指定大小的释放操作：

/include/linux/kernel.h

```
12   #define  free(x)  free_s((x), 0)
```

当正是释放内存块时，关闭了中断，仍然是为了避免进程切换造成数据结构的不一致。free_s 的代码如下：

/lib/malloc.c

```
174
175 /*
176 * Here is the free routine. If you know the size of the object that you
177 * are freeing, then free_s() will use that information to speed up the
178 * search for the bucket descriptor.
179 *
180 * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
181 */
/*
* 下面是释放子程序。如果你知道释放对象的大小，则 free_s() 将使用该信息加速
* 搜寻对应桶描述符的速度。
*
* 我们将定义一个宏，使得 "free(x)" 成为 "free_s(x, 0)"。
*/
182 void free_s(void *obj, int size)
183 {
184   void *page;
185   struct _bucket_dir *bdir;
186   struct bucket_desc *bdesc, *prev;
187
188  /* Calculate what page this object lives in */
```

```
    /* 计算该对象所在的页面 */
189   page = (void *) ((unsigned long) obj & 0xfffff000);
190 /* Now search the buckets looking for that page */
    /* 现在搜索存储桶目录项所链接的桶描述符，寻找该页面 */
191 for (bdir = bucket_dir; bdir->size; bdir++) {
192       prev = 0;                                   // 如果是 0，则搜索到的桶是链表的第一个桶
193   /* If size is zero then this conditional is always false */
       /* 如果参数 size 是 0，则下面条件肯定是 false */
194     if (bdir->size < size)
195         continue;
196     for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
197       if (bdesc->page == page)
198           goto found;                             // 退出搜索
199       prev = bdesc;
200     }
201   }
202   panic( "Bad address passed to kernel free_s()");                    // 如果搜索失败
203 found:
204   cli();              /* To avoid race conditions */ /* 为了避免竞争条件 */
205    *((void **)obj) = bdesc->freeptr;                                  // 链入空闲块链表
206    bdesc->freeptr = obj;
207    bdesc->refcnt--;
208    if (bdesc->refcnt == 0) {
209 /*
210 * We need to make sure that prev is still accurate. It
211 * may not be, if someone rudely interrupted us....
212 */
/*
* 我们需要确信 prev 仍然是正确的，若某程序粗鲁地中断了我们
* 就有可能不是了。
*/
213         if ((prev && (prev->next != bdesc)) || (!prev && (bdir->chain != bdesc)))
214
215             for (prev = bdir->chain; prev; prev = prev->next)
216                 if (prev->next == bdesc)
217                     break;
218         if (prev)
219             prev->next = bdesc->next;
220         else {                              // prev 如果是 0，则搜索到的桶是链表的第一个桶
221             if (bdir->chain != bdesc)
```

```
222                     panic( "malloc bucket chains corrupted");
223                 bdir->chain = bdesc->next;
224               }
225           free_page((unsigned long) bdesc->page);                    // 释放对应内存页面
226           bdesc->next = free_bucket_desc;                            // 将桶描述符链入空闲桶描述符链表
227           free_bucket_desc = bdesc;
228       }         /*  208    if (bdesc->refcnt == 0)*/
229     sti();                                      // 开中断
230     return;
231 }
232
```

9 内存使用情况统计

通过前面的介绍，对于 linux0.11 内存管理机制有了整体的了解。有时候需要对主内存块的使用情况作统计。统计的项目有：当前主内存块还有多少空闲页面；分配了的物理页面在 4G 的虚拟内存空间中的分布情况。

对于第一个项目，只需要对 主内存块映射数组 mem_map 的空闲项进行统计。对于第二个项目，需要对 4G 的虚存空间进行查询，必须查页目录表和页表。通过页目录表查每个页表的所有项，通过判断每个页表项的有效位 (P) 统计每个页表中映射有多少个物理页面。第一节曾经谈到内核将 16M 的物理内存映射到第 0 ～ 3 个页表上从而实现对物理内存的控制，在这里这几个页表是没有必要查询的，因为始终是页表的 1024 项都映射了物理页面。但是 linux0.11 也对 2、3 两个页表进行了统计，只跳过了 0、1 两个页表。以下便是统计函数 calc_mem(void)

/mm/memory.c

```
// 计算主内存块中空闲页面数并统计每个页表中映射了的物理页面数。
413 void calc_mem(void)
414 {
415    int i,j,k,free=0;
416    long * pg_tbl;
417
// 扫描映射数组 mem_map[]，统计主内存块中的空闲页面数并显示。
418    for(i=0 ; i<PAGING_PAGES ; i++)
419            if (!mem_map[i]) free++;
420    printk( "%d pages free (of %d)\n\r",free,PAGING_PAGES);
// 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。
421   for(i=2 ; i<1024 ; i++) {
422      if (1&pg_dir[i]) {
423         pg_tbl=(long *) (0xfffff000 & pg_dir[i]);
424         for(j=k=0 ; j<1024 ; j++)
425            if (pg_tbl[j]&1)
426                  k++;
427         printk( "Pg-dir[%d] uses %d pages\n",i,k);
428       }
429   }
430 }
```