[**内存管理之2.配合fork**](https://www.cnblogs.com/joey-hua/p/5598451.html)

【版权所有，转载请注明出处。出处：<http://www.cnblogs.com/joey-hua/p/5598451.html> 】

 在上一篇的fork函数中，首先一上来就调用get\_free\_page为新任务的数据结构申请一页内存，在memory.c中：

[?](https://www.cnblogs.com/joey-hua/p/5598451.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | /\*  \* 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。  \*/  //// 取空闲页面。如果已经没有可用内存了，则返回0。  // 输入：%1(ax=0) - 0；%2(LOW\_MEM)；%3(cx=PAGING PAGES)；%4(edi=mem\_map+PAGING\_PAGES-1)。  // 输出：返回%0(ax=页面起始地址)。  // 上面%4 寄存器实际指向mem\_map[]内存字节图的最后一个字节。本函数从字节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（其内存映像字节为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页面，但并没有映射到某个进程的线性地址去。后面  // 的put\_page()函数就是用来作映射的。  unsigned long  get\_free\_page (void)  {    register unsigned long \_\_res asm ("ax");      \_\_asm\_\_ ("std ; repne ; scasb\n\t"    // 方向位置位，将al(0)与对应每个页面的(di)内容比较，         "jne 1f\n\t"                                         // 如果没有等于0 的字节，则跳转结束（返回0）。         "movb $1,1(%%edi)\n\t"                   // 将对应页面的内存映像位置1。         "sall $12,%%ecx\n\t"                     // 页面数\*4K = 相对页面起始地址。         "addl %2,%%ecx\n\t"                      // 再加上低端内存地址，即获得页面实际物理起始地址。         "movl %%ecx,%%edx\n\t"               // 将页面实际起始地址??edx 寄存器。         "movl $1024,%%ecx\n\t"               // 寄存器ecx 置计数值1024。         "leal 4092(%%edx),%%edi\n\t"     // 将4092+edx 的位置??edi(该页面的末端)。         "rep ; stosl\n\t"                                    // 将edi 所指内存清零（反方向，也即将该页面清零）。         "movl %%edx,%%eax\n"                 // 将页面起始地址??eax（返回值）。  "1:": "=a" (\_\_res): "" (0), "i" (LOW\_MEM), "c" (PAGING\_PAGES), "D" (mem\_map + PAGING\_PAGES - 1):"di", "cx",         "dx");    return \_\_res;                                     // 返回空闲页面地址（如果无空闲也则返回0）。  } |

上面有几个指令比较陌生，先介绍repne scasb，其对应的等价指令是：

[?](https://www.cnblogs.com/joey-hua/p/5598451.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | scans:inc edi      dec ecx      je loopdone      cmp byte [edi-1],al      jne scans  loopdone: |

sall $12,%eax表示将%eax的值左移12位，相当于eax=eax\*4096.

STOSL指令相当于将EAX中的值保存到ES:EDI指向的地址中。

所以第一句指令的意思是把al即%0的值0与di内容比较（倒序），edi为mem\_map+PAGING\_PAGES-1，即内存映射数组的最后一个可分页的下标内容，如果有等于0的字节表示还未使用，就将对应页面的内存映像位置1.

然后把ecx，此时不再是PAGING\_PAGES，乘以4096得到相对页面的起始地址，再加上LOW\_MEM得到页面实际物理起始地址。然后把这整页内存清0.最后返回这个页面的起始地址。

接下来看最关键的copy\_page\_tables函数：

[?](https://www.cnblogs.com/joey-hua/p/5598451.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | // 刷新页变换高速缓冲宏函数。  // 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。  // 下面eax = 0，是页目录的基址。  #define invalidate() \  \_\_asm\_\_( "movl %%eax,%%cr3":: "a" (0))    /\*  \* 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）。  \*/  //// 复制指定线性地址和长度（页表个数）内存对应的页目录项和页表，从而被复制的页目录和  //// 页表对应的原物理内存区被共享使用。  // 复制指定地址和长度的内存对应的页目录项和页表项。需申请页面来存放新页表，原内存区被共享；  // 此后两个进程将共享内存区，直到有一个进程执行写操作时，才分配新的内存页（写时复制机制）。  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;    // 源地址和目的地址都需要是在4Mb 的内存边界地址上。否则出错，死机。    if ((from & 0x3fffff) || (to & 0x3fffff))      panic ("copy\_page\_tables called with wrong alignment");  // 取得源地址和目的地址的目录项(from\_dir 和to\_dir)。参见对115 句的注释。    from\_dir = (unsigned long \*) ((from >> 20) & 0xffc);    /\* \_pg\_dir = 0 \*/    to\_dir = (unsigned long \*) ((to >> 20) & 0xffc);  // 计算要复制的内存块占用的页表数（也即目录项数）。    size = ((unsigned) (size + 0x3fffff)) >> 22;  // 下面开始对每个占用的页表依次进行复制操作。    for (; size-- > 0; from\_dir++, to\_dir++)      {  // 如果目的目录项指定的页表已经存在(P=1)，则出错，死机。        if (1 & \*to\_dir)      panic ("copy\_page\_tables: already exist");  // 如果此源目录项未被使用，则不用复制对应页表，跳过。        if (!(1 & \*from\_dir))      continue;  // 取当前源目录项中页表的地址??from\_page\_table。        from\_page\_table = (unsigned long \*) (0xfffff000 & \*from\_dir);  // 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。        if (!(to\_page\_table = (unsigned long \*) get\_free\_page ()))      return -1;      /\* Out of memory, see freeing \*/  // 设置目的目录项信息。7 是标志信息，表示(Usr, R/W, Present)。        \*to\_dir = ((unsigned long) to\_page\_table) | 7;  // 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  // 复制1 个页表中的所有1024 页面。        nr = (from == 0) ? 0xA0 : 1024;  // 对于当前页表，开始复制指定数目nr 个内存页面。        for (; nr-- > 0; from\_page\_table++, to\_page\_table++)      {        this\_page = \*from\_page\_table;         // 取源页表项内容。        if (!(1 & this\_page))                             // 如果当前源页面没有使用，则不用复制。          continue;  // 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，  // 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。)        this\_page &= ~2;        \*to\_page\_table = this\_page;               // 将该页表项复制到目的页表中。  // 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  // 页面号，并以它为索引在页面映射数组相应项中增加引用次数。而对于位于1MB 以下的页面，说明  // 是内核页面，因此不需要对mem\_map[]进行设置。因为mem\_map[]仅用于管理主内存区中的页面使用  // 情况。因此，对于内核移动到任务0 中并且调用fork()创建任务1 时（用于运行init()），由于此  //时  // 复制的页面还仍然都在内核代码区域，因此以下判断中的语句不会执行。只有当调用fork()的父进程  // 代码处于主内存区（页面位置大于1MB）时才会执行。这种情况需要在进程调用了execve()，装载并  // 执行了新程序代码时才会出现。        if (this\_page > LOW\_MEM)          {  // 下面这句的含义是令源页表项所指内存页也为只读。因为现在开始有两个进程共用内存区了。  // 若其中一个内存需要进行写操作，则可以通过页异常的写保护处理，为执行写操作的进程分配  // 一页新的空闲页面，也即进行写时复制的操作。            \*from\_page\_table = this\_page;     // 令源页表项也只读。            this\_page -= LOW\_MEM;            this\_page >>= 12;            mem\_map[this\_page]++;          }      }      }    invalidate ();        // 刷新页变换高速缓冲。    return 0;  } |

记得从fork传递过来的三个参数依次是old\_data\_base，new\_data\_base，data\_limit。其中old\_data\_base是原进程局部描述符表中数据段的基地址（线性地址空间），new\_data\_base为新进程在线性地址空间中的基地址（任务号\*64MB），data\_limit为原进程的局部描述符表中数据段描述符中的段限长。

首先取源地址和目的地址的页目录项，因为一页内存为4K即4096，所以4096对应的是一个页表项，由于一个页表有1024个表项，所以一个页表为1024\*4096=4194304，又由于一个完整的页表对应的是一个页目录项，所以**页目录号**即为地址除以4194304（即右移22位）。因为每项占4个字节，并且由于页目录是从物理地址0开始（head.s），因此实际的页目录项指针=页目录号\*4（即左移2）。和0xffc（4092）相与表示不能超出1024个页目录项的范围。

紧接着计算限长的页目录项数，也即所占页表数，(size+4M)/4M。

然后用一个for循环依次复制每个占用的页表，首先取源目录项中的页表地址0xfffff000 & \*from\_dir，根据PDE的结构，12-31位为页表基地址，0-11位为各种属性。所以用0xfffff000清除低12位，获取高20位的页表基址。

接下来为**目的页表**申请一页空白内存，此页表的起始地址存在to\_page\_table中，并置前三位为1.再将这个地址值赋值给目的页目录项。

然后又用一个for循环复制以from\_page\_table为页表起始地址的一整个页表的页表项内容，首先取第一个源页表项的内容\*from\_page\_table，其实就是某个页的地址和一些属性。然后将该页表项内容this\_page赋值给\*to\_page\_table。

后面一小段代码是设置只读。

最后一句为刷新页变换高速缓冲，没什么好说的。

上面的函数执行如果出错，则会调用free\_page\_tables来释放申请的内存：

[?](https://www.cnblogs.com/joey-hua/p/5598451.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | /\*  \* 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。  \*/  //// 根据指定的线性地址和限长（页表个数），释放对应内存页表所指定的内存块并置表项空闲。  // 页目录位于物理地址0 开始处，共1024 项，占4K 字节。每个目录项指定一个页表。  // 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。  // 每个页表项对应一页物理内存（4K）。目录项和页表项的大小均为4 个字节。  // 参数：from - 起始基地址；size - 释放的长度。  int  free\_page\_tables (unsigned long from, unsigned long size)  {    unsigned long \*pg\_table;    unsigned long \*dir, nr;      if (from & 0x3fffff)                                  // 要释放内存块的地址需以4M 为边界。                                                                    //不能<4M，小于4M就等于本身，大于4M就等于0      panic ("free\_page\_tables called with wrong alignment");    if (!from)                                                    // 出错，试图释放内核和缓冲所占空间。      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。(size+4M)/4M  //一个页是4KB，一整个页表有1024个页，所以4KB\*1024=4M就是一整个页表所对应的size容量  //然后一整个页表对应的是一个页目录项    size = (size + 0x3fffff) >> 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，并且由于页目录是从  // 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。    dir = (unsigned long \*) ((from >> 20) & 0xffc); /\* \_pg\_dir = 0 \*/    for (; size-- > 0; dir++)      {                                                               // size 现在是需要被释放内存的目录项数。        if (!(1 & \*dir))                                      // 如果该目录项无效(P 位=0)，则继续。      continue;                                               // 目录项的位0(P 位)表示对应页表是否存在。        pg\_table = (unsigned long \*) (0xfffff000 & \*dir); // 取目录项中页表地址。        for (nr = 0; nr < 1024; nr++)      {                                                               // 每个页表有1024 个页项。        if (1 & \*pg\_table)                                // 若该页表项有效(P 位=1)，则释放对应内存页。          free\_page (0xfffff000 & \*pg\_table);        \*pg\_table = 0;                                        // 该页表项内容清零。        pg\_table++;                                           // 指向页表中下一项。      }        free\_page (0xfffff000 & \*dir);            // 释放该页表所占内存页面。但由于页表在                                                                      // 物理地址1M 以内，所以这句什么都不做。        \*dir = 0;                                             // 对相应页表的目录项清零。      }    invalidate ();                                            // 刷新页变换高速缓冲。    return 0;  } |

这个函数和上面的函数类似，首先计算所占页目录项数，然后计算起始目录项地址。

然后用一个for循环先取到目录项中的页表地址，再用一个for循环把页表中的1024个页项清空，这里又用到一个函数free\_page：

[?](https://www.cnblogs.com/joey-hua/p/5598451.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | /\*  \* 释放物理地址'addr'开始的一页内存。用于函数'free\_page\_tables()'。  \*/  //// 释放物理地址addr 开始的一页面内存。  // 1MB 以下的内存空间用于内核程序和缓冲，不作为分配页面的内存空间。  //a = i--;//先a = i ; 然后 i = i - 1;  void  free\_page (unsigned long addr)  {    if (addr < LOW\_MEM)      return;                                         // 如果物理地址addr 小于内存低端（1MB），则返回。    if (addr >= HIGH\_MEMORY)   // 如果物理地址addr>=内存最高端，则显示出错信息。      panic ("trying to free nonexistent page");    addr -= LOW\_MEM;                      // 物理地址减去低端内存位置，再除以4KB，得页面号。    addr >>= 12;    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。    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。    panic ("trying to free free page");  } |

这个函数是释放一页内存，首先得到页面号，然后把内存映射数组对应的下标的内容减1.比较简单。

所以free\_page (0xfffff000 & \*pg\_table);的含义是先取页表项的内容，也就是对应的某一页内存的地址，然后释放这一页内存。

释放完这一页内存后，就把该页表项内容清零\*pg\_table=0.

接着再释放该页表所占的内存页面（4K），最后释放该页目录项的内容。

至此分析结束！