[**加载可执行二进制文件之3.exec**](https://www.cnblogs.com/joey-hua/p/5638591.html)

最后剩下最核心的函数do\_execve了，由于这里为了简单起见我不分析shell命令的情况，

|  |
| --- |
| /\*  \* 'do\_execve()'函数执行一个新程序。  \*/  //// execve()系统中断调用函数。加载并执行子进程（其它程序）。  // 该函数系统中断调用(int 0x80)功能号\_\_NR\_execve 调用的函数。  // 参数：eip - 指向堆栈中调用系统中断的程序代码指针eip 处，参见kernel/system\_call.s 程序  // 开始部分的说明；tmp - 系统中断调用本函数时的返回地址，无用；  // filename - 被执行程序文件名；argv - 命令行参数指针数组；envp - 环境变量指针数组。  // 返回：如果调用成功，则不返回；否则设置出错号，并返回-1。  int  do\_execve (unsigned long \*eip, long tmp, char \*filename,         char \*\*argv, char \*\*envp)  {    struct m\_inode \*inode;    // 内存中I 节点指针结构变量。    struct buffer\_head \*bh;   // 高速缓存块头指针。    struct exec ex;       // 执行文件头部数据结构变量。    unsigned long page[MAX\_ARG\_PAGES];    // 参数和环境字符串空间的页面指针数组。    int i, argc, envc;    int e\_uid, e\_gid;     // 有效用户id 和有效组id。    int retval;           // 返回值。    int sh\_bang = 0;      // 控制是否需要执行脚本处理代码。  // 参数和环境字符串空间中的偏移指针，初始化为指向该空间的最后一个长字处。    unsigned long p = PAGE\_SIZE \* MAX\_ARG\_PAGES - 4;    // eip[1]中是原代码段寄存器cs，其中的选择符不可以是内核段选择符，也即内核不能调用本函数。    if ((0xffff & eip[1]) != 0x000f)      panic ("execve called from supervisor mode");  // 初始化参数和环境串空间的页面指针数组（表）。    for (i = 0; i < MAX\_ARG\_PAGES; i++)    /\* clear page-table \*/      page[i] = 0;  // 取可执行文件的对应i 节点号。    if (!(inode = namei (filename)))  /\* get executables inode \*/      return -ENOENT;  // 计算参数个数和环境变量个数。    argc = count (argv);    envc = count (envp);    // 执行文件必须是常规文件。若不是常规文件则置出错返回码，跳转到exec\_error2(第347 行)。  restart\_interp:    if (!S\_ISREG (inode->i\_mode))      {               /\* must be regular file \*/        retval = -EACCES;        goto exec\_error2;      }  // 检查被执行文件的执行权限。根据其属性(对应i 节点的uid 和gid)，看本进程是否有权执行它。    i = inode->i\_mode;             // 取文件属性字段值。      // 如果文件的设置用户ID 标志（set-user-id）置位的话，则后面执行进程的有效用户ID（euid）就    // 设置为文件的用户ID，否则设置成当前进程的euid。这里将该值暂时保存在e\_uid 变量中。    // 如果文件的设置组ID 标志（set-group-id）置位的话，则执行进程的有效组ID（egid）就设置为    // 文件的组ID。否则设置成当前进程的egid。    e\_uid = (i & S\_ISUID) ? inode->i\_uid : current->euid;    e\_gid = (i & S\_ISGID) ? inode->i\_gid : current->egid;    // 如果文件属于运行进程的用户，则把文件属性字右移6 位，则最低3 位是文件宿主的访问权限标志。    // 否则的话如果文件与运行进程的用户属于同组，则使属性字最低3 位是文件组用户的访问权限标志。    // 否则属性字最低3 位是其他用户访问该文件的权限。    if (current->euid == inode->i\_uid)      i >>= 6;    else if (current->egid == inode->i\_gid)      i >>= 3;    // 如果上面相应用户没有执行权并且其他用户也没有任何权限，并且不是超级用户，则表明该文件不    // 能被执行。于是置不可执行出错码，跳转到exec\_error2 处去处理。    if (!(i & 1) && !((inode->i\_mode & 0111) && suser ()))      {        retval = -ENOEXEC;        goto exec\_error2;      }  // 读取执行文件的第一块数据到高速缓冲区，若出错则置出错码，跳转到exec\_error2 处去处理。    if (!(bh = bread (inode->i\_dev, inode->i\_zone[0])))      {        retval = -EACCES;        goto exec\_error2;      }  // 下面对执行文件的头结构数据进行处理，首先让ex 指向执行头部分的数据结构。    ex = \*((struct exec \*) bh->b\_data);    /\* read exec-header \*//\* 读取执行头部分 \*/  ...  //shell  ...  // 释放该缓冲区。    brelse (bh);  // 下面对执行头信息进行处理。  // 对于下列情况，将不执行程序：如果执行文件不是需求页可执行文件(ZMAGIC)、或者代码重定位部分  // 长度a\_trsize 不等于0、或者数据重定位信息长度不等于0、或者代码段+数据段+堆段长度超过50MB、  // 或者i 节点表明的该执行文件长度小于代码段+数据段+符号表长度+执行头部分长度的总和。    if (N\_MAGIC (ex) != ZMAGIC || ex.a\_trsize || ex.a\_drsize ||        ex.a\_text + ex.a\_data + ex.a\_bss > 0x3000000 ||        inode->i\_size < ex.a\_text + ex.a\_data + ex.a\_syms + N\_TXTOFF (ex))      {        retval = -ENOEXEC;        goto exec\_error2;      }  // 如果执行文件执行头部分长度不等于一个内存块大小（1024 字节），也不能执行。转exec\_error2。    if (N\_TXTOFF (ex) != BLOCK\_SIZE)      {        printk ("%s: N\_TXTOFF != BLOCK\_SIZE. See a.out.h.", filename);        retval = -ENOEXEC;        goto exec\_error2;      }  // 如果sh\_bang 标志没有设置，则复制指定个数的环境变量字符串和参数到参数和环境空间中。  // 若sh\_bang 标志已经设置，则表明是将运行脚本程序，此时环境变量页面已经复制，无须再复制。    if (!sh\_bang)      {        p = copy\_strings (envc, envp, page, p, 0);        p = copy\_strings (argc, argv, page, p, 0);  // 如果p=0，则表示环境变量与参数空间页面已经被占满，容纳不下了。转至出错处理处。        if (!p)      {        retval = -ENOMEM;        goto exec\_error2;      }      }  /\* OK, This is the point of no return \*/  /\* OK，下面开始就没有返回的地方了 \*/  // 如果原程序也是一个执行程序，则释放其i 节点，并让进程executable 字段指向新程序i 节点。    if (current->executable)      iput (current->executable);    current->executable = inode;  // 清复位所有信号处理句柄。但对于SIG\_IGN 句柄不能复位，因此在322 与323 行之间需添加一条  // if 语句：if (current->sa[I].sa\_handler != SIG\_IGN)。这是源代码中的一个bug。    for (i = 0; i < 32; i++)      current->sigaction[i].sa\_handler = NULL;  // 根据执行时关闭(close\_on\_exec)文件句柄位图标志，关闭指定的打开文件，并复位该标志。    for (i = 0; i < NR\_OPEN; i++)      if ((current->close\_on\_exec >> i) & 1)        sys\_close (i);    current->close\_on\_exec = 0;  // 根据指定的基地址和限长，释放原来程序代码段和数据段所对应的内存页表指定的内存块及页表本身。    // 此时被执行程序没有占用主内存区任何页面。在执行时会引起内存管理程序执行缺页处理而为其申请    // 内存页面，并把程序读入内存。    free\_page\_tables (get\_base (current->ldt[1]), get\_limit (0x0f));    free\_page\_tables (get\_base (current->ldt[2]), get\_limit (0x17));  // 如果“上次任务使用了协处理器”指向的是当前进程，则将其置空，并复位使用了协处理器的标志。    if (last\_task\_used\_math == current)      last\_task\_used\_math = NULL;    current->used\_math = 0;  // 根据a\_text 修改局部表中描述符基址和段限长，并将参数和环境空间页面放置在数据段末端。  // 执行下面语句之后，p 此时是以数据段起始处为原点的偏移值，仍指向参数和环境空间数据开始处，  // 也即转换成为堆栈的指针。    p += change\_ldt (ex.a\_text, page) - MAX\_ARG\_PAGES \* PAGE\_SIZE;  // create\_tables()在新用户堆栈中创建环境和参数变量指针表，并返回该堆栈指针。    p = (unsigned long) create\_tables ((char \*) p, argc, envc);  // 修改当前进程各字段为新执行程序的信息。令进程代码段尾值字段end\_code = a\_text；令进程数据  // 段尾字段end\_data = a\_data + a\_text；令进程堆结尾字段brk = a\_text + a\_data + a\_bss。    current->brk = ex.a\_bss +      (current->end\_data = ex.a\_data + (current->end\_code = ex.a\_text));  // 设置进程堆栈开始字段为堆栈指针所在的页面，并重新设置进程的有效用户id 和有效组id。    current->start\_stack = p & 0xfffff000;    current->euid = e\_uid;    current->egid = e\_gid;  // 初始化一页bss 段数据，全为零。    i = ex.a\_text + ex.a\_data;    while (i & 0xfff)      put\_fs\_byte (0, (char \*) (i++));  // 将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将堆栈指针替换  // 为新执行程序的堆栈指针。返回指令将弹出这些堆栈数据并使得CPU 去执行新的执行程序，因此不会  // 返回到原调用系统中断的程序中去了。    eip[0] = ex.a\_entry;      /\* eip, magic happens :-) \*//\* eip，魔法起作用了 \*/    eip[3] = p;           /\* stack pointer \*//\* esp，堆栈指针 \*/    return 0;  exec\_error2:    iput (inode);  exec\_error1:    for (i = 0; i < MAX\_ARG\_PAGES; i++)      free\_page (page[i]);    return (retval);  } |

尽管删掉很大一部分，但代码还是很长。不过没有关系，核心代码还是一小部分，大部分是判断性的代码。判断性的代码就不做分析了，仔细看也是能看懂。

注意bh = bread (inode->i\_dev, inode->i\_zone[0]))首先读取可执行文件的第一块数据到高速缓冲区，紧接着ex = \*((struct exec \*) bh->b\_data);把b\_data数据复制到ex中，这说明文件的b\_data型数据就是exec结构。

后面又是一堆对ex的判断。

接着调用copy\_strings，这时p指向参数和环境空间的已使用的地址处。

后面又是一堆给当前进程current赋值的操作。

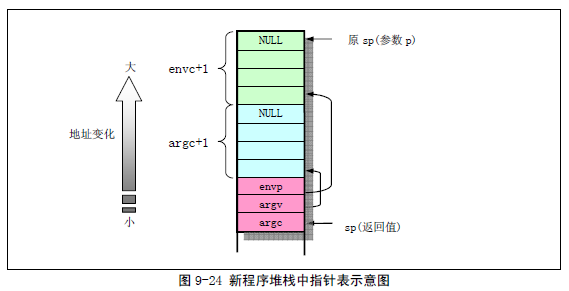
然后是两次free\_page\_tables释放LDT的代码段和数据段。

然后调用change\_ldt设置当前进程的LDT，这时p指向的位置之前分析过了。

接下来分析create\_tables：

|  |
| --- |
| /\*  \* create\_tables()函数在新用户内存中解析环境变量和参数字符串，由此  \* 创建指针表，并将它们的地址放到"堆栈"上，然后返回新栈的指针值。  \*/  //// 在新用户堆栈中创建环境和参数变量指针表。  // 参数：p - 以数据段为起点的参数和环境信息偏移指针；argc - 参数个数；envc -环境变量数。  // 返回：堆栈指针。  static unsigned long \*  create\_tables (char \*p, int argc, int envc)  {    unsigned long \*argv, \*envp;    unsigned long \*sp;    // 堆栈指针是以4 字节（1 节）为边界寻址的，因此这里让sp 为4 的整数倍。    sp = (unsigned long \*) (0xfffffffc & (unsigned long) p);  // sp 向下移动，空出环境参数占用的空间个数，并让环境参数指针envp 指向该处。    sp -= envc + 1;    envp = sp;  // sp 向下移动，空出命令行参数指针占用的空间个数，并让argv 指针指向该处。  // 下面指针加1，sp 将递增指针宽度字节值。    sp -= argc + 1;    argv = sp;  // 将环境参数指针envp 和命令行参数指针以及命令行参数个数压入堆栈。    put\_fs\_long ((unsigned long) envp, --sp);    put\_fs\_long ((unsigned long) argv, --sp);    put\_fs\_long ((unsigned long) argc, --sp);  // 将命令行各参数指针放入前面空出来的相应地方，最后放置一个NULL 指针。    while (argc-- > 0)      {        put\_fs\_long ((unsigned long) p, argv++);        while (get\_fs\_byte (p++)) /\* nothing \*/ ; // p 指针前移4 字节。      }    put\_fs\_long (0, argv);  // 将环境变量各指针放入前面空出来的相应地方，最后放置一个NULL 指针。    while (envc-- > 0)      {        put\_fs\_long ((unsigned long) p, envp++);        while (get\_fs\_byte (p++)) /\* nothing \*/ ;      }    put\_fs\_long (0, envp);    return sp;            // 返回构造的当前新堆栈指针。  } |

create\_tables()函数用于根据给定的当前堆栈指针值p 以及参数变量个数值argc 和环境变量个数  
envc，在新的程序堆栈中创建环境和参数变量指针表，并返回此时的堆栈指针值sp。创建完毕后堆栈指  
针表的形式见下图9-24 所示。



注意这里有三条连续的put\_fs\_long函数调用，这里的压入堆栈并不是真的压栈，而是以压栈的方式存数据。

然后是两个循环放置p对应的字节值，注意这里是p++，因为之前拷贝的时候是--p。

数据填充完成后返回sp。

接着后面给当前进程的start\_stack赋值，赋值p所在的页面。

最后值得注意的是：

|  |
| --- |
| // 将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将堆栈指针替换  // 为新执行程序的堆栈指针。返回指令将弹出这些堆栈数据并使得CPU 去执行新的执行程序，因此不会  // 返回到原调用系统中断的程序中去了。    eip[0] = ex.a\_entry;      /\* eip, magic happens :-) \*//\* eip，魔法起作用了 \*/    eip[3] = p;           /\* stack pointer \*//\* esp，堆栈指针 \*/ |

do\_execve是在system\_call.s中调用的：

|  |
| --- |
| #### 这是sys\_execve()系统调用。取中断调用程序的代码指针作为参数调用C 函数do\_execve()。  # do\_execve()在(fs/exec.c,182)。  .align 2  \_sys\_execve:  lea EIP(%esp),%eax  pushl %eax  call \_do\_execve  addl $4,%esp                        # 丢弃调用时压入栈的EIP 值。  ret |

可以观察到，这个EIP是调用\_system\_call之前压入的，所以do\_execve的eip就是这个EIP：

|  |
| --- |
| /\*  \* 0(%esp) - %eax  \* 4(%esp) - %ebx  \* 8(%esp) - %ecx  \* C(%esp) - %edx  \* 10(%esp) - %fs  \* 14(%esp) - %es  \* 18(%esp) - %ds  \* 1C(%esp) - %eip  \* 20(%esp) - %cs  \* 24(%esp) - %eflags  \* 28(%esp) - %oldesp  \* 2C(%esp) - %oldss  \*/ |

所以eip[0]就是%eip，赋值了程序的入口地址，eip[3]就是%oldesp，赋值了p的值。

至此exec.c分析结束！