```
1  /*
2   *  linux/fs/exec.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * #!-checking implemented by tytso.
9   */
   /*
    * #!开始的脚本程序的检测代码部分是由 tytso 实现的。
    */
10
11 /*
12  * Demand-loading implemented 01.12.91 - no need to read anything but
13  * the header into memory. The inode of the executable is put into
14  * "current->executable", and page faults do the actual loading. Clean.
15  *
16  * Once more I can proudly say that linux stood up to being changed: it
17  * was less than 2 hours work to get demand-loading completely implemented.
18  */
   /*
    * 需求时加载实现于 1991.12.1 - 只需将执行文件头部读进内存而无须将整个
    * 执行文件都加载进内存。执行文件的 i 节点被放在当前进程的可执行字段中
    * "current->executable"，页异常会进行执行文件的实际加载操作。这很完美。
    *
    * 我可以再一次自豪地说，linux 经得起修改：只用了不到 2 小时的工作时间就
    * 完全实现了需求加载处理。
    */
19
20 #include <signal.h>       // 信号头文件。定义信号符号常量，信号结构及信号操作函数原型。
21 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
22 #include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
23 #include <sys/stat.h>     // 文件状态头文件。含有文件状态结构 stat{}和符号常量等。
24 #include <a.out.h>        // a.out 头文件。定义了 a.out 执行文件格式和一些宏。
25
26 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file、m_inode）等。
27 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
28 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
29 #include <linux/mm.h>     // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
30 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
31
32 extern int sys_exit(int exit_code);     // 退出程序系统调用。
33 extern int sys_close(int fd);           // 关闭文件系统调用。
34
35 /*
36  * MAX_ARG_PAGES defines the number of pages allocated for arguments
37  * and envelope for the new program. 32 should suffice, this gives
38  * a maximum env+arg of 128kB !
39  */
   /*
    * MAX_ARG_PAGES 定义了为新程序分配的给参数和环境变量使用的最大内存页数。
```

```
 * 32 页内存应该足够了，这使得环境和参数（env+arg）空间的总和达到 128kB!
 */

40 #define MAX_ARG_PAGES 32
41
   //// 使用库文件系统调用。
   // 参数：library - 库文件名。
   // 为进程选择一个库文件，并替换进程当前库文件 i 节点字段值为这里指定库文件名的 i 节点
   // 指针。如果 library 指针为空，则把进程当前的库文件释放掉。
   // 返回：成功返回 0，否则返回出错码。
42 int sys_uselib(const char * library)
43 {
44         struct m_inode * inode;
45         unsigned long base;
46
   // 首先判断当前进程是否普通进程。这是通过查看当前进程的空间长度来做到的。因为普通进
   // 程的空间长度被设置为 TASK_SIZE（64MB）。因此若进程逻辑地址空间长度不等于 TASK_SIZE
   // 则返回出错码（无效参数）。否则取库文件 i 节点 inode。若库文件名指针空，则设置 inode
   // 等于 NULL。
47         if (get_limit(0x17) != TASK_SIZE)
48                 return -EINVAL;
49         if (library) {
50                 if (!(inode=namei(library)))            /* get library inode */
51                         return -ENOENT;                 /* 取库文件 i 节点 */
52         } else
53                 inode = NULL;
54 /* we should check filetypes (headers etc), but we don't */
   /* 我们应该检查一下文件类型（如头部信息等），但是我们还没有这样做。 */
   // 然后放回进程原库文件 i 节点，并预置进程库 i 节点字段为空。接着取得进程的库代码所在
   // 位置，并释放原库代码的页表和所占用的内存页面。最后让进程库 i 节点字段指向新库 i 节
   // 点，并返回 0（成功）。
55         iput(current->library);
56         current->library = NULL;
57         base = get_base(current->ldt[2]);
58         base += LIBRARY_OFFSET;
59         free_page_tables(base,LIBRARY_SIZE);
60         current->library = inode;
61         return 0;
62 }
63
64 /*
65  * create_tables() parses the env- and arg-strings in new user
66  * memory and creates the pointer tables from them, and puts their
67  * addresses on the "stack", returning the new stack pointer value.
68  */
   /*
    * create_tables()函数在新任务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"栈"上，然后返回新栈的指针值。
    */
   //// 在新任务栈中创建参数和环境变量指针表。
   // 参数：p -数据段中参数和环境信息偏移指针；argc - 参数个数；envc - 环境变量个数。
   // 返回：栈指针值。
69 static unsigned long * create_tables(char * p,int argc,int envc)
```

```
{
        unsigned long *argv,*envp;
        unsigned long * sp;

// 栈指针是以 4 字节（1 节）为边界进行寻址的，因此这里需让 sp 为 4 的整数倍值。此时 sp
// 位于参数环境表的末端。然后我们先把 sp 向下（低地址方向）移动，在栈中空出环境变量
// 指针占用的空间，并让环境变量指针 envp 指向该处。多空出的一个位置用于在最后存放一
// 个 NULL 值。下面指针加 1，sp 将递增指针宽度字节值（4 字节）。再把 sp 向下移动，空出
// 命令行参数指针占用的空间，并让 argv 指针指向该处。同样，多空处的一个位置用于存放
// 一个 NULL 值。此时 sp 指向参数指针块的起始处，我们将环境参数块指针 envp 和命令行参
// 数块指针以及命令行参数个数值分别压入栈中。
        sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
        sp -= envc+1;                                   // 即 sp = sp - (envc+1);
        envp = sp;
        sp -= argc+1;
        argv = sp;
        put_fs_long((unsigned long)envp,--sp);
        put_fs_long((unsigned long)argv,--sp);
        put_fs_long((unsigned long)argc,--sp);
// 再将命令行各参数指针和环境变量各指针分别放入前面空出来的相应地方，最后分别放置一
// 个 NULL 指针。
        while (argc-->0) {
                put_fs_long((unsigned long) p,argv++);
                while (get_fs_byte(p++)) /* nothing */ ; // p 指针指向下一个参数串。
        }
        put_fs_long(0,argv);
        while (envc-->0) {
                put_fs_long((unsigned long) p,envp++);
                while (get_fs_byte(p++)) /* nothing */ ; // p 指针指向下一个参数串。
        }
        put_fs_long(0,envp);
        return sp;                              // 返回构造的当前新栈指针。
}

/*
 * count() counts the number of arguments/envelopes
 */
/*
 * count()函数计算命令行参数/环境变量的个数。
 */
//// 计算参数个数。
// 参数：argv - 参数指针数组，最后一个指针项是 NULL。
// 统计参数指针数组中指针的个数。关于函数参数传递指针的指针的作用，请参见程序
// kernel/sched.c 中第 171 行前的注释。
// 返回：参数个数。
static int count(char ** argv)
{
        int i=0;
        char ** tmp;

        if (tmp = argv)
                while (get_fs_long((unsigned long *) (tmp++)))
                        i++;
```

```
106
107         return i;
108 }
109
110 /*
111  * 'copy_string()' copies argument/envelope strings from user
112  * memory to free pages in kernel mem. These are in a format ready
113  * to be put directly into the top of new user memory.
114  *
115  * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
116  * whether the string and the string array are from user or kernel segments:
117  *
118  * from_kmem     argv *        argv **
119  *    0          user space    user space
120  *    1          kernel space  user space
121  *    2          kernel space  kernel space
122  *
123  * We do this by playing games with the fs segment register.  Since it
124  * it is expensive to load a segment register, we try to avoid calling
125  * set_fs() unless we absolutely have to.
126  */
    /*
     * 'copy_string()'函数从用户内存空间拷贝参数/环境字符串到内核空闲页面中。
     * 这些已具有直接放到新用户内存中的格式。
     *
     * 由 TYT(Tytso)于 1991.11.24 日修改，增加了 from_kmem 参数，该参数指明了
     * 字符串或字符串数组是来自用户段还是内核段。
     *
     * from_kmem   指针 argv *     字符串 argv **
     *    0          用户空间       用户空间
     *    1          内核空间       用户空间
     *    2          内核空间       内核空间
     *
     * 我们是通过巧妙处理 fs 段寄存器来操作的。由于加载一个段寄存器代价太高，
     * 所以我们尽量避免调用 set_fs()，除非实在必要。
     */
    //// 复制指定个数的参数字符串到参数和环境空间中。
    // 参数：argc - 欲添加的参数个数；argv - 参数指针数组；page - 参数和环境空间页面指针
    // 数组。p -参数表空间中偏移指针，始终指向已复制串的头部；from_kmem -字符串来源标志。
    // 在 do_execve()函数中，p 初始化为指向参数表(128kB)空间的最后一个长字处，参数字符串
    // 是以堆栈操作方式逆向往其中复制存放的。因此 p 指针会随着复制信息的增加而逐渐减小，
    // 并始终指向参数字符串的头部。字符串来源标志 from_kmem 应该是 TYT 为了给 execve()增添
    // 执行脚本文件的功能而新加的参数。当没有运行脚本文件的功能时，所有参数字符串都在用
    // 户数据空间中。
    // 返回：参数和环境空间当前头部指针。若出错则返回 0。
127 static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
128                 unsigned long p, int from_kmem)
129 {
130         char *tmp, *pag;
131         int len, offset = 0;
132         unsigned long old_fs, new_fs;
133
    // 首先取当前段寄存器 ds（指向内核数据段）和 fs 值，分别保存到变量 new_fs 和 old_fs 中。
```

```
    // 如果字符串和字符串数组（指针）来自内核空间，则设置 fs 段寄存器指向内核数据段。
134         if (!p)
135                 return 0;       /* bullet-proofing */  /* 偏移指针验证 */
136         new_fs = get_ds();
137         old_fs = get_fs();
138         if (from_kmem==2)           // 若串及其指针在内核空间则设置 fs 指向内核空间。
139                 set_fs(new_fs);
    // 然后循环处理各个参数，从最后一个参数逆向开始复制，复制到指定偏移地址处。在循环中，
    // 首先取需要复制的当前字符串指针。如果字符串在用户空间而字符串数组（字符串指针）在
    // 内核空间，则设置 fs 段寄存器指向内核数据段（ds）。并在内核数据空间中取了字符串指针
    // tmp 之后就立刻恢复 fs 段寄存器原值（fs 再指回用户空间）。否则不用修改 fs 值而直接从
    // 用户空间取字符串指针到 tmp。
140         while (argc-- > 0) {
141                 if (from_kmem == 1)      // 若串指针在内核空间，则 fs 指向内核空间。
142                         set_fs(new_fs);
143                 if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
144                         panic("argc is wrong");
145                 if (from_kmem == 1)      // 若串指针在内核空间，则 fs 指回用户空间。
146                         set_fs(old_fs);
    // 然后从用户空间取该字符串，并计算该参数字符串长度 len。 此后 tmp 指向该字符串末端。
    // 如果该字符串长度超过此时参数和环境空间中还剩余的空闲长度，则空间不够了。于是恢复
    // fs 段寄存器值（如果被改变的话）并返回 0。不过因为参数和环境空间留有 128KB，所以通
    // 常不可能发生这种情况。
147                 len=0;                          /* remember zero-padding */
148                 do {                            /* 我们知道串是以 NULL 字节结尾的 */
149                         len++;
150                 } while (get_fs_byte(tmp++));
151                 if (p-len < 0) {                /* this shouldn't happen - 128kB */
152                         set_fs(old_fs);         /* 不会发生-因为有 128kB 的空间 */
153                         return 0;
154                 }
    // 接着我们逆向逐个字符地把字符串复制到参数和环境空间末端处。在循环复制字符串的字符
    // 过程中，我们首先要判断参数和环境空间中相应位置处是否已经有内存页面。如果还没有就
    // 先为其申请 1 页内存页面。偏移量 offset 被用作为在一个页面中的当前指针偏移值。 因为
    // 刚开始执行本函数时，偏移变量 offset 被初始化为 0，所以(offset-1 < 0)肯定成立而使得
    // offset 重新被设置为当前 p 指针在页面范围内的偏移值。
155                 while (len) {
156                         --p; --tmp; --len;
157                         if (--offset < 0) {
158                                 offset = p % PAGE_SIZE;
159                                 if (from_kmem==2) // 若串在内核空间则 fs 指回用户空间。
160                                         set_fs(old_fs);
    // 如果当前偏移值 p 所在的串空间页面指针数组项 page[p/PAGE_SIZE] ==0，表示此时 p 指针
    // 所处的空间内存页面还不存在，则需申请一空闲内存页，并将该页面指针填入指针数组，同
    // 时也使页面指针 pag 指向该新页面。若申请不到空闲页面则返回 0。
161                                 if (!(pag = (char *) page[p/PAGE_SIZE]) &&
162                                     !(pag = (char *) page[p/PAGE_SIZE] =
163                                       (unsigned long *) get_free_page()))
164                                         return 0;
165                                 if (from_kmem==2) // 若串在内核空间则 fs 指向内核空间。
166                                         set_fs(new_fs);
167
168                         }
```

```
    // 然后从 fs 段中复制字符串的 1 字节到参数和环境空间内存页面 pag 的 offset 处。
169                         *(pag + offset) = get_fs_byte(tmp);
170                 }
171         }
    // 如果字符串和字符串数组在内核空间，则恢复 fs 段寄存器原值。最后，返回参数和环境空
    // 间中已复制参数的头部偏移值。
172         if (from_kmem==2)
173                 set_fs(old_fs);
174         return p;
175 }
176
    //// 修改任务的局部描述符表内容。
    // 修改局部描述符表 LDT 中描述符的段基址和段限长，并将参数和环境空间页面放置在数据段
    // 末端。
    // 参数：text_size - 执行文件头部中 a_text 字段给出的代码段长度值；
    // page - 参数和环境空间页面指针数组。
    // 返回：数据段限长值（64MB）。
177 static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
178 {
179         unsigned long code_limit,data_limit,code_base,data_base;
180         int i;
181
    // 首先把代码和数据段长度均设置为 64MB。 然后取当前进程局部描述符表代码段描述符中代
    // 码段基址。代码段基址与数据段基址相同。再使用这些新值重新设置局部表中代码段和数据
    // 段描述符中的基址和段限长。这里请注意，由于被加载的新程序的代码和数据段基址与原程
    // 序的相同，因此没有必要再重复去设置它们，即第 186 和 188 行上的两条设置段基址的语句
    // 多余，可省略。
182         code_limit = TASK_SIZE;
183         data_limit = TASK_SIZE;
184         code_base = get_base(current->ldt[1]);  // include/linux/sched.h，第 226 行。
185         data_base = code_base;
186         set_base(current->ldt[1],code_base);
187         set_limit(current->ldt[1],code_limit);
188         set_base(current->ldt[2],data_base);
189         set_limit(current->ldt[2],data_limit);
190 /* make sure fs points to the NEW data segment */
    /* 要确信 fs 段寄存器已指向新的数据段 */
    // fs 段寄存器中放入局部表数据段描述符的选择符(0x17)。即默认情况下 fs 都指向任务数据段。
191         __asm__("pushl $0x17\n\tpop %%fs"::);
    // 然后将参数和环境空间已存放数据的页面（最多有 MAX_ARG_PAGES 页，128kB）放到数据段末
    // 端。 方法是从进程空间库代码位置开始处逆向一页一页地放。库文件代码占用进程空间最后
    // 4MB。函数 put_dirty_page()用于把物理页面映射到进程逻辑空间中。在 mm/memory.c 中。
192         data_base += data_limit - LIBRARY_SIZE;
193         for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
194                 data_base -= PAGE_SIZE;
195                 if (page[i])                            // 若该页面存在，就放置该页面。
196                         put_dirty_page(page[i],data_base);
197         }
198         return data_limit;                              // 最后返回数据段限长（64MB）。
199 }
200
201 /*
202  * 'do_execve()' executes a new program.
```

```
203  *
204  * NOTE! We leave 4MB free at the top of the data-area for a loadable
205  * library.
206  */
     /*
      * '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（其他程序）。
     // 该函数是系统中断调用（int 0x80）功能号__NR_execve 调用的函数。函数的参数是进入系统
     // 调用处理过程后直到调用本系统调用处理过程（system_call.s 第 200 行）和调用本函数之前
     // （system_call.s，第 203 行）逐步压入栈中的值。这些值包括：
     // ① 第 86--88 行入堆的 edx、ecx 和 ebx 寄存器值，分别对应**envp、**argv 和*filename；
     // ② 第 94 行调用 sys_call_table 中 sys_execve 函数（指针）时压入栈的函数返回地址（tmp）；
     // ③ 第 202 行在调用本函数 do_execve 前入栈的指向栈中调用系统中断的程序代码指针 eip。
     // 参数：
     // eip - 调用系统中断的程序代码指针，参见 kernel/system_call.s 程序开始部分的说明；
     // tmp - 系统中断中在调用_sys_execve 时的返回地址，无用；
     // filename - 被执行程序文件名指针；
     // argv - 命令行参数指针数组的指针；
     // envp - 环境变量指针数组的指针。
     // 返回：如果调用成功，则不返回；否则设置出错号，并返回-1。
207  int do_execve(unsigned long * eip,long tmp,char * filename,
208          char ** argv, char ** envp)
209  {
210          struct m_inode * inode;
211          struct buffer_head * bh;
212          struct exec ex;
213          unsigned long page[MAX_ARG_PAGES];          // 参数和环境串空间页面指针数组。
214          int i,argc,envc;
215          int e_uid, e_gid;                           // 有效用户 ID 和有效组 ID。
216          int retval;
217          int sh_bang = 0;                            // 控制是否需要执行脚本程序。
218          unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4; // p 指向参数和环境空间的最后部。
219
     // 在正式设置执行文件的运行环境之前，让我们先干些杂事。内核准备了 128KB（32 个页面）
     // 空间来存放化执行文件的命令行参数和环境字符串。上行把 p 初始设置成位于 128KB 空间的
     // 最后 1 个长字处。在初始参数和环境空间的操作过程中，p 将用来指明在 128KB 空间中的当
     // 前位置。
     // 另外，参数 eip[1]是调用本次系统调用的原用户程序代码段寄存器 CS 值，其中的段选择符
     // 当然必须是当前任务的代码段选择符（0x000f）。若不是该值，那么 CS 只能会是内核代码
     // 段的选择符 0x0008。 但这是绝对不允许的，因为内核代码是常驻内存而不能被替换掉的。
     // 因此下面根据 eip[1] 的值确认是否符合正常情况。然后再初始化 128KB 的参数和环境串空
     // 间，把所有字节清零，并取出执行文件的 i 节点。再根据函数参数分别计算出命令行参数和
     // 环境字符串的个数 argc 和 envc。另外，执行文件必须是常规文件。
220          if ((0xffff & eip[1]) != 0x000f)
221                  panic("execve called from supervisor mode");
222          for (i=0 ; i<MAX_ARG_PAGES ; i++)       /* clear page-table */
223                  page[i]=0;
224          if (!(inode=namei(filename)))           /* get executables inode */
225                  return -ENOENT;
226          argc = count(argv);                     // 命令行参数个数。
227          envc = count(envp);                     // 环境字符串变量个数。
228
```

```
229 restart_interp:
230         if (!S_ISREG(inode->i_mode)) {    /* must be regular file */
231                 retval = -EACCES;
232                 goto exec_error2;         //若不是常规文件则置出错码，跳转到 376 行。
233         }
    // 下面检查当前进程是否有权运行指定的执行文件。即根据执行文件 i 节点中的属性，看看本
    // 进程是否有权执行它。在把执行文件 i 节点的属性字段值取到 i 中后，我们首先查看属性中
    // 是否设置了“设置-用户-ID”（set-user_id）标志 和“设置-组-ID”（set-group-id）标
    // 志。这两个标志主要是让一般用户能够执行特权用户（如超级用户 root）的程序，例如改变
    // 密码的程序 passwd 等。如果 set-user-id 标志置位，则后面执行进程的有效用户 ID（euid）
    // 就设置成执行文件的用户 ID，否则设置成当前进程的 euid。如果执行文件 set-group-id 被
    // 置位的话，则执行进程的有效组 ID（egid）就设置为执行文件的组 ID。否则设置成当前进程
    // 的 egid。这里暂时把这两个判断出来的值保存在变量 e_uid 和 e_gid 中。
234         i = inode->i_mode;
235         e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
236         e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;

    // 现在根据进程的 euid 和 egid 和执行文件的访问属性进行比较。如果执行文件属于运行进程
    // 的用户，则把文件属性值 i 右移 6 位，此时其最低 3 位是文件宿主的访问权限标志。否则的
    // 话如果执行文件与当前进程的用户属于同组，则使属性值最低 3 位是执行文件组用户的访问
    // 权限标志。否则此时属性字最低 3 位就是其他用户访问该执行文件的权限。
    // 然后我们根据属性字 i 的最低 3 比特值来判断当前进程是否有权限运行这个执行文件。如果
    // 选出的相应用户没有运行改文件的权力（位 0 是执行权限），并且其他用户也没有任何权限
    // 或者当前进程用户不是超级用户，则表明当前进程没有权力运行这个执行文件。于是置不可
    // 执行出错码，并跳转到 exec_error2 处去作退出处理。
237         if (current->euid == inode->i_uid)
238                 i >>= 6;
239         else if (in_group_p(inode->i_gid))
240                 i >>= 3;
241         if (!(i & 1) &&
242             !((inode->i_mode & 0111) && suser())) {
243                 retval = -ENOEXEC;
244                 goto exec_error2;
245         }
    // 程序执行到这里，说明当前进程有运行指定执行文件的权限。因此从这里开始我们需要取出
    // 执行文件头部数据并根据其中的信息来分析设置运行环境，或者运行另一个 shell 程序来执
    // 行脚本程序。首先读取执行文件第 1 块数据到高速缓冲块中。 并复制缓冲块数据到 ex 中。
    // 如果执行文件开始的两个字节是字符'#!'，则说明执行文件是一个脚本文本文件。如果想运
    // 行脚本文件，我们就需要执行脚本文件的解释程序（例如 shell 程序）。通常脚本文件的第
    // 一行文本为“#! /bin/bash”。它指明了运行脚本文件需要的解释程序。运行方法是从脚本
    // 文件第 1 行（带字符'#!'）中取出其中的解释程序名及后面的参数（若有的话），然后将这
    // 些参数和脚本文件名放进执行文件（此时是解释程序）的命令行参数空间中。在这之前我们
    // 当然需要先把函数指定的原有命令行参数和环境字符串放到 128KB 空间中，而这里建立起来
    // 的命令行参数则放到它们前面位置处（因为是逆向放置）。最后让内核执行脚本文件的解释
    // 程序。下面就是在设置好解释程序的脚本文件名等参数后，取出解释程序的 i 节点并跳转到
    // 229 行去执行解释程序。由于我们需要跳转到执行过的代码 229 行去，因此在下面确认并处
    // 理了脚本文件之后需要设置一个禁止再次执行下面的脚本处理代码标志 sh_bang 。在后面的
    // 代码中该标志也用来表示我们已经设置好执行文件的命令行参数，不要重复设置。
246         if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
247                 retval = -EACCES;
248                 goto exec_error2;
249         }
250         ex = *((struct exec *) bh->b_data);     /* read exec-header */
```

```
251         if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
252                 /*
253                  * This section does the #! interpretation.
254                  * Sorta complicated, but hopefully it will work.  -TYT
255                  */
                    /*
                     * 这部分处理对'#!'的解释，有些复杂，但希望能工作。-TYT
                     */
256
257                 char buf[128], *cp, *interp, *i_name, *i_arg;
258                 unsigned long old_fs;
259
    // 从这里开始，我们从脚本文件中提取解释程序名及其参数，并把解释程序名、解释程序的参数
    // 和脚本文件名组合放入环境参数块中。首先复制脚本文件头 1 行字符'#!'后面的字符串到 buf
    // 中，其中含有脚本解释程序名（例如/bin/sh），也可能还包含解释程序的几个参数。然后对
    // buf 中的内容进行处理。删除开始的空格、制表符。
260                 strncpy(buf, bh->b_data+2, 127);
261                 brelse(bh);
262                 iput(inode);                        // 释放缓冲块并放回脚本文件 i 节点。
263                 buf[127] = '\0';
264                 if (cp = strchr(buf, '\n')) {
265                         *cp = '\0';         // 第 1 个换行符换成 NULL 并去掉空格制表符。
266                         for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
267                 }
268                 if (!cp || *cp == '\0') {           // 若该行没有其他内容，则出错。
269                         retval = -ENOEXEC;          /* No interpreter name found */
270                         goto exec_error1;           /* 没有找到脚本解释程序名 */
271                 }
    // 此时我们得到了开头是脚本解释程序名的一行内容（字符串）。下面分析该行。首先取第一个
    // 字符串，它应该是解释程序名，此时 i_name 指向该名称。若解释程序名后还有字符，则它们应
    // 该是解释程序的参数串，于是令 i_arg 指向该串。
272                 interp = i_name = cp;
273                 i_arg = 0;
274                 for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
275                         if (*cp == '/')
276                                 i_name = cp+1;
277                 }
278                 if (*cp) {
279                         *cp++ = '\0';               // 解释程序名尾添加 NULL 字符。
280                         i_arg = cp;                 // i_arg 指向解释程序参数。
281                 }
282                 /*
283                  * OK, we've parsed out the interpreter name and
284                  * (optional) argument.
285                  */
                    /*
                     * OK，我们已经解析出解释程序的文件名以及(可选的)参数。
                     */
    // 现在我们要把上面解析出来的解释程序名 i_name 及其参数 i_arg 和脚本文件名作为解释程
    // 序的参数放进环境和参数块中。不过首先我们需要把函数提供的原来一些参数和环境字符串
    // 先放进去，然后再放这里解析出来的。例如对于命令行参数来说，如果原来的参数是"-arg1
    //  -arg2"、解释程序名是"bash"、其参数是"-iarg1 -iarg2"、脚本文件名（即原来的执行文
    // 件名）是"example.sh"，那么在放入这里的参数之后，新的命令行类似于这样：
```

```
//          “bash -iarg1 -iarg2 example.sh -arg1 -arg2”
// 这里我们把 sh_bang 标志置上，然后把函数参数提供的原有参数和环境字符串放入到空间中。
// 环境字符串和参数个数分别是 envc 和 argc-1 个。少复制的一个原有参数是原来的执行文件
// 名，即这里的脚本文件名。[[?? 可以看出，实际上我们不需要去另行处理脚本文件名，即这
// 里完全可以复制 argc 个参数，包括原来执行文件名（即现在的脚本文件名）。因为它位于同
// 一个位置上 ]]。注意！这里指针 p 随着复制信息增加而逐渐向小地址方向移动，因此这两个
// 复制串函数执行完后，环境参数串信息块位于程序命令行参数串信息块的上方，并且 p 指向
// 程序的第 1 个参数串。copy_strings()最后一个参数（0）指明参数字符串在用户空间。
286                 if (sh_bang++ == 0) {
287                         p = copy_strings(envc, envp, page, p, 0);
288                         p = copy_strings(--argc, argv+1, page, p, 0);
289                 }
290                 /*
291                  * Splice in (1) the interpreter's name for argv[0]
292                  *           (2) (optional) argument to interpreter
293                  *           (3) filename of shell script
294                  *
295                  * This is done in reverse order, because of how the
296                  * user environment and arguments are stored.
297                  */
                    /*
                     * 拼接 (1) argv[0]中放解释程序的名称
                     *      (2) (可选的)解释程序的参数
                     *      (3) 脚本程序的名称
                     *
                     * 这是以逆序进行处理的，是由于用户环境和参数的存放方式造成的。
                     */
// 接着我们逆向复制脚本文件名、解释程序的参数和解释程序文件名到参数和环境空间中。
// 若出错，则置出错码，跳转到 exec_error1。另外，由于本函数参数提供的脚本文件名
// filename 在用户空间，但这里赋予 copy_strings()的脚本文件名的指针在内核空间，因
// 此这个复制字符串函数的最后一个参数（字符串来源标志）需要被设置成 1。若字符串在
// 内核空间，则 copy_strings()的最后一个参数要设置成 2，如下面的第 301、304 行。
298                 p = copy_strings(1, &filename, page, p, 1);
299                 argc++;
300                 if (i_arg) {                        // 复制解释程序的多个参数。
301                         p = copy_strings(1, &i_arg, page, p, 2);
302                         argc++;
303                 }
304                 p = copy_strings(1, &i_name, page, p, 2);
305                 argc++;
306                 if (!p) {
307                         retval = -ENOMEM;
308                         goto exec_error1;
309                 }
310                 /*
311                  * OK, now restart the process with the interpreter's inode.
312                  */
                    /*
                     * OK，现在使用解释程序的 i 节点重启进程。
                     */
// 最后我们取得解释程序的 i 节点指针，然后跳转到 204 行去执行解释程序。为了获得解释程
// 序的 i 节点，我们需要使用 namei()函数，但是该函数所使用的参数（文件名）是从用户数
// 据空间得到的，即从段寄存器 fs 所指空间中取得。因此在调用 namei()函数之前我们需要
```

```
    // 先临时让 fs 指向内核数据空间，以让函数能从内核空间得到解释程序名，并在 namei()
    // 返回后恢复 fs 的默认设置。因此这里我们先临时保存原 fs 段寄存器（原指向用户数据段）
    // 的值，将其设置成指向内核数据段，然后取解释程序的 i 节点。之后再恢复 fs 的原值。并
    // 跳转到 restart_interp（204 行）处重新处理新的执行文件 -- 脚本文件的解释程序。
313                 old_fs = get_fs();
314                 set_fs(get_ds());
315                 if (!(inode=namei(interp))) {        /* get executables inode */
316                         set_fs(old_fs);              /* 取得解释程序的 i 节点 */
317                         retval = -ENOENT;
318                         goto exec_error1;
319                 }
320                 set_fs(old_fs);
321                 goto restart_interp;
322         }

    // 此时缓冲块中的执行文件头结构数据已经复制到了 ex 中。 于是先释放该缓冲块，并开始对
    // ex 中的执行头信息进行判断处理。对于 Linux 0.12 内核来说，它仅支持 ZMAGIC 执行文件格
    // 式，并且执行文件代码都从逻辑地址 0 开始执行，因此不支持含有代码或数据重定位信息的
    // 执行文件。当然，如果执行文件实在太大或者执行文件残缺不全，那么我们也不能运行它。
    // 因此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
    // 码和数据重定位部分长度不等于 0、或者(代码段+数据段+堆) 长度超过 50MB、或者执行文件
    // 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。
323         brelse(bh);
324         if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
325                 ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
326                 inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
327                 retval = -ENOEXEC;
328                 goto exec_error2;
329         }
    // 另外，如果执行文件中代码开始处没有位于 1 个页面（1024 字节）边界处，则也不能执行。
    // 因为需求页（Demand paging）技术要求加载执行文件内容时以页面为单位，因此要求执行
    // 文件映像中代码和数据都从页面边界处开始。
330         if (N_TXTOFF(ex) != BLOCK_SIZE) {
331                 printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
332                 retval = -ENOEXEC;
333                 goto exec_error2;
334         }
    // 如果 sh_bang 标志没有设置，则复制指定个数的命令行参数和环境字符串到参数和环境空间
    // 中。若 sh_bang 标志已经设置，则表明是将运行脚本解释程序，此时环境变量页面已经复制，
    // 无须再复制。同样，若 sh_bang 没有置位而需要复制的话，那么此时指针 p 随着复制信息增
    // 加而逐渐向小地址方向移动，因此这两个复制串函数执行完后，环境参数串信息块位于程序
    // 参数串信息块的上方，并且 p 指向程序的第 1 个参数串。事实上，p 是 128KB 参数和环境空
    // 间中的偏移值。因此如果 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。
335         if (!sh_bang) {
336                 p = copy_strings(envc,envp,page,p,0);
337                 p = copy_strings(argc,argv,page,p,0);
338                 if (!p) {
339                         retval = -ENOMEM;
340                         goto exec_error2;
341                 }
342         }
343 /* OK, This is the point of no return */
344 /* note that current->library stays unchanged by an exec */
```

```
    /* OK，下面开始就没有返回的地方了 */
    // 前面我们针对函数参数提供的信息对需要运行执行文件的命令行参数和环境空间进行了设置，
    // 但还没有为执行文件做过什么实质性的工作，即还没有做过为执行文件初始化进程任务结构
    // 信息、建立页表等工作。现在我们就来做这些工作。由于执行文件直接使用当前进程的“躯
    // 壳”，即当前进程将被改造成执行文件的进程，因此我们需要首先释放当前进程占用的某些
    // 系统资源，包括关闭指定的已打开文件、占用的页表和内存页面等。然后根据执行文件头结
    // 构信息修改当前进程使用的局部描述符表 LDT 中描述符的内容，重新设置代码段和数据段描
    // 述符的限长，再利用前面处理得到的 e_uid 和 e_gid 等信息来设置进程任务结构中相关的字
    // 段。最后把执行本次系统调用程序的返回地址 eip[]指向执行文件中代码的起始位置处。这
    // 样当本系统调用退出返回后就会去运行新执行文件的代码了。注意，虽然此时新执行文件代
    // 码和数据还没有从文件中加载到内存中，但其参数和环境块已经在 copy_strings() 中使用
    // get_free_page()分配了物理内存页来保存数据，并在 change_ldt()函数中使用 put_page()
    // 到了进程逻辑空间的末端处。另外，在 create_tables()中也会由于在用户栈上存放参数
    // 和环境指针表而引起缺页异常，从而内存管理程序也会就此为用户栈空间映射物理内存页。
    //
    // 这里我们首先放回进程原执行程序的 i 节点，并且让进程 executable 字段指向新执行文件
    // 的 i 节点。然后复位原进程的所有信号处理句柄，但对于 SIG_IGN 句柄无须复位。再根据设
    // 定的执行时关闭文件句柄（close_on_exec）位图标志，关闭指定的打开文件并复位该标志。
345         if (current->executable)
346                 iput(current->executable);
347         current->executable = inode;
348         current->signal = 0;
349         for (i=0 ; i<32 ; i++) {
350                 current->sigaction[i].sa_mask = 0;
351                 current->sigaction[i].sa_flags = 0;
352                 if (current->sigaction[i].sa_handler != SIG_IGN)
353                         current->sigaction[i].sa_handler = NULL;
354         }
355         for (i=0 ; i<NR_OPEN ; i++)
356                 if ((current->close_on_exec>>i)&1)
357                         sys_close(i);
358         current->close_on_exec = 0;
    // 然后根据当前进程指定的基地址和限长，释放原来程序的代码段和数据段所对应的内存页表
    // 指定的物理内存页面及页表本身。此时新执行文件并没有占用主内存区任何页面，因此在处
    // 理器真正运行新执行文件代码时就会引起缺页异常中断，此时内存管理程序即会执行缺页处
    // 理而为新执行文件申请内存页面和设置相关页表项，并且把相关执行文件页面读入内存中。
    // 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。
359         free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
360         free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
361         if (last_task_used_math == current)
362                 last_task_used_math = NULL;
363         current->used_math = 0;
    // 然后我们根据新执行文件头结构中的代码长度字段 a_text 的值修改局部表中描述符基址和
    // 段限长，并将 128KB 的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
    // 更改成以数据段起始处为原点的偏移值，但仍指向参数和环境空间数据开始处，即已转换成
    // 为栈指针值。然后调用内部函数 create_tables() 在栈空间中创建环境和参数变量指针表，
    // 供程序的 main()作为参数使用，并返回该栈指针。
364         p += change_ldt(ex.a_text,page);
365         p -= LIBRARY_SIZE + MAX_ARG_PAGES*PAGE_SIZE;
366         p = (unsigned long) create_tables((char *)p,argc,envc);

    // 接着再修改进程各字段值为新执行文件的信息。即令进程任务结构代码尾字段 end_code 等
```

```
    // 于执行文件的代码段长度 a_text；数据尾字段 end_data 等于执行文件的代码段长度加数
    // 据段长度（a_data + a_text）；并令进程堆结尾字段 brk = a_text + a_data + a_bss。
    // brk 用于指明进程当前数据段（包括未初始化数据部分）末端位置，供内核为进程分配内存
    // 时指定分配开始位置。然后设置进程栈开始字段为栈指针所在页面，并重新设置进程的有效
    // 用户 id 和有效组 id。
367         current->brk = ex.a_bss +
368                 (current->end_data = ex.a_data +
369                 (current->end_code = ex.a_text));
370         current->start_stack = p & 0xfffff000;
371         current->suid = current->euid = e_uid;
372         current->sgid = current->egid = e_gid;
    // 最后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，并将栈
    // 指针替换为新执行文件的栈指针。此后返回指令将弹出这些栈数据并使得 CPU 去执行新执行
    // 文件，因此不会返回到原调用系统中断的程序中去了。
373         eip[0] = ex.a_entry;    /* eip, magic happens :-) */  /* eip，魔法起作用了*/
374         eip[3] = p;             /* stack pointer */           /* esp，堆栈指针 */
375         return 0;
376 exec_error2:
377         iput(inode);                        // 放回 i 节点。
378 exec_error1:
379         for (i=0 ; i<MAX_ARG_PAGES ; i++)
380                 free_page(page[i]);         // 释放存放参数和环境串的内存页面。
381         return(retval);                     // 返回出错码。
382 }
383
```