Linux 下 ELF 可执行文件载入（内核部分）源码分析

引子

一个程序是死的，当我们在 shell 里面敲入其名字，它就“活”了。这里面发生的事情实在太多。作为 programmer，可以用现有的工具来看看到底从死到活发生了什么。以最简单的 Hello World 程序为例。

```
[wzhou@dcmp10 ~]$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
2.2.5, dynamically linked (uses shared libs), not stripped
```

由于我用的是动态链接，所以“file”命令报告的是 ELF 格式的 dynamiclly linked executable。

```
[wzhou@dcmp10 ~]$ ldd hello
      libc.so.6 => /lib/tls/libc.so.6 (0x0077f000)
      /lib/ld-linux.so.2 (0x00766000)
```

ldd 命令报告“Hello World”程序引用了两个动态库。

1. /lib/tls/libc.so.6，这自然是 C 库，因为 printf()属于标准 C 库。
2. /lib/ld-linux.so.2，这是动态链接器，可执行程序中依赖的共享库就是由它负责载入的。

在 strace 的监控下运行一下

```
[wzhou@dcmp10 ~]$ strace ./hello
execve("./hello", ["./hello"], [/* 22 vars */]) = 0
uname({sys="Linux", node="dcmp10", ...}) = 0
brk(0)                                  = 0x9ae3000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=76373, ...}) = 0
old_mmap(NULL, 76373, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fed000
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0 ?y\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1454835, ...}) = 0
old_mmap(0x77f000, 1215644, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x77f000
old_mmap(0x8a2000,                 16384,                 PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x123000) = 0x8a2000
old_mmap(0x8a6000,                 7324,                  PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x8a6000
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7fec000
```

```
mprotect(0x8a2000, 4096, PROT_READ)    = 0
mprotect(0x77b000, 4096, PROT_READ)    = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7fec940, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
munmap(0xb7fed000, 76373)            = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 9), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7fff000
write(1, "Hello, World!\n", 14Hello, World!
)       = 14
fstat64(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 9), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7ffe000
read(0,
"\n", 1024)                  = 1
munmap(0xb7fff000, 4096)             = 0
exit_group(0)                       = ?
[wzhou@dcmp10 ~]$
```

Strace 截获了所有的系统调用。再来看一下程序运行后的空间分配。

```
[wzhou@dcmp10 ~]$ ./hello &
[2] 3518
[wzhou@dcmp10 ~]$ Hello, World!


[2]+  Stopped                 ./hello
[wzhou@dcmp10 ~]$ cat /proc/3518/maps
00766000-0077b000 r-xp 00000000 fd:00 492674    /lib/ld-2.3.4.so
0077b000-0077c000 r--p 00014000 fd:00 492674    /lib/ld-2.3.4.so
0077c000-0077d000 rw-p 00015000 fd:00 492674    /lib/ld-2.3.4.so
0077f000-008a2000 r-xp 00000000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a2000-008a3000 r--p 00123000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a3000-008a6000 rw-p 00124000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a6000-008a8000 rw-p 008a6000 00:00 0
08048000-08049000 r-xp 00000000 fd:01 378914    /home/wzhou/hello
08049000-0804a000 rw-p 00000000 fd:01 378914    /home/wzhou/hello
b7fec000-b7fed000 rw-p b7fec000 00:00 0
b7ffe000-b8000000 rw-p b7ffe000 00:00 0
bfeca000-c0000000 rw-p bfeca000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
[wzhou@dcmp10 ~]$
```

上面的内存映射由两个阶段完成

```
00766000-0077b000 r-xp 00000000 fd:00 492674    /lib/ld-2.3.4.so
```

```
0077b000-0077c000 r--p 00014000 fd:00 492674    /lib/ld-2.3.4.so
0077c000-0077d000 rw-p 00015000 fd:00 492674    /lib/ld-2.3.4.so

08048000-08049000 r-xp 00000000 fd:01 378914    /home/wzhou/hello
08049000-0804a000 rw-p 00000000 fd:01 378914    /home/wzhou/hello
```

上面的内存分配是在

```
execve("./hello", ["./hello"], [/* 22 vars */]) = 0
```

这行系统调用时完成。而

```
0077f000-008a2000 r-xp 00000000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a2000-008a3000 r--p 00123000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a3000-008a6000 rw-p 00124000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a6000-008a8000 rw-p 008a6000 00:00 0
```

由动态链接器/lib/ld-2.3.4.so 完成。

由于前者是完全在 kernel 中完成，所以 strace tool 不能跟踪到，但后者却完全在用户态执行，所以可以从 strace 的输出上得到印证。

比如

```
old_mmap(0x77f000, 1215644, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x77f000
old_mmap(0x8a2000, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x123000) = 0x8a2000
old_mmap(0x8a6000, 7324, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x8a6000

old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fec000
```

首先映射了如上四大块内存，然后通过调用 mprotect 系统调用来修改已映射页的属性。

```
mprotect(0x8a2000, 4096, PROT_READ)    = 0
mprotect(0x77b000, 4096, PROT_READ)    = 0
```

这个过程非常复杂，大致分为如下几步（以在 shell (bash shell)里敲入如下命令为例）。

$./hello （执行该命令）

1. bash shell “fork”一个与自身一模一样的 process。

该 process 执行系统调用 execve()，也就是你看到的 strace 输出的第一行

execve("./hello", ["./hello"], [/* 22 vars */]) = 0。由于无论是读取文件，还是文件映射(File Mapping)都发生在 Kernel 里，所以 strace 不能监控到这一步的任何东西。**本文要分析的即是该过程**。

2. 系统调用 execve()以后，该 process 的真正入口被设置成动态链接器的入口（如果是静态链接的话，就比较简单，设置的入口就是该可执行文件 hello 的入口）。当该新 process 返回到用户态（也就是从 execve 执行完后从内核态返回到用户态，不是对该函数调用的返回。因为只有该函数调用失败才会有返回，成功的话，就进入了另一个“世界”），控制转移到动态链接器（/lib/ld-2.3.4.so）。

3. 动态链接器依次载入该可执行文件要用到的动态库。而且这是一个递归的过程，即要载入的共享库可能又用到其他的共享库。由于这些都发生在用户态，所以都被 strace 跟踪到。这部分的代码在 GNU C 库(glibc)中，有待以后分析。（如果有人分析好了，那请告诉我，我并不是代码狂，象爱看小说一样爱看代码。不过，最好不要给我看的分析是目前在网上多如牛毛的所谓粗略分析，其详细度应该与我分析系统调用 execve()差相仿佛）。

4. 动态链接器把控制转移到可执行文件的入口。这以后的事请看我的另一篇分析文章《main 之前与之后》。

源码解析

这里列出 ELF 可执行文件载入时用到的相关主要函数及对应的完整注释。如果你在读源码，希望对你有所帮助。如果发现我有理解有误的地方，希望指出。如果你对 ELF 格式，Unix programming 等一点概念也没有，那我想你可能会失望。

sys_execve源码

```
/*
 * sys_execve() executes a new program.
 */
asmlinkage int sys_execve(struct pt_regs regs)
{
int error;
char * filename;

filename = getname((char __user *) regs.ebx);
error = PTR_ERR(filename);
if (IS_ERR(filename))
    goto out;
error = do_execve(filename,
       (char __user * __user *) regs.ecx,
       (char __user * __user *) regs.edx,
       &regs);
if (error == 0) {
    task_lock(current);
    current->ptrace &= ~PT_DTRACE;
    task_unlock(current);
    /* Make sure we don't return using sysenter.. */
    set_thread_flag(TIF_IRET);
}
putname(filename);
out:
return error;
}
```

sys_execve注释

```
/*
 * sys_execve() executes a new program.
 */
asmlinkage int sys_execve(struct pt_regs regs)
{
    int error;
    char * filename;

    filename = getname((char __user *) regs.ebx);
    error = PTR_ERR(filename);
    if (IS_ERR(filename))
        goto out;
```

getname()完成从用户空间拷贝可执行文件的文件名到内核空间，在该函数中会分配 kernel space,所以在该函数退出前要"putname(filename)"。

```
    error = do_execve(filename,
            (char __user * __user *) regs.ecx,
            (char __user * __user *) regs.edx,
            &regs);
```

见 do_execve（）的分析。

```
    if (error == 0) {
```

成功 execute 该程序。

```
        task_lock(current);
```

为修改当前 process 的 task_struct 中的成员，要先占有该锁。

```
        current->ptrace &= ~PT_DTRACE;
```

这里清除掉单步执行的标志。如果是正常的启动某个程序，当然这行代码等于什么都没做（因为本来该标志就是 clear 的）；但如果是某个 debugger "execute"该可执行文件，则就有效了。作用是什么？

```
        task_unlock(current);
```

释放锁。

```
        /* Make sure we don't return using sysenter.. */
        set_thread_flag(TIF_IRET);
```

设置 flag，不要用 2.6 内核引入的快速系统调用返回方式返回。

```
    }
```

```
    putname(filename);
释放在上面 getname()中为存放可执行文件的 filename 申请的内核空间。
out:
    return error;
}
```

do_execve源码

```
/*
 * sys_execve() executes a new program.
 */
int do_execve(char * filename,
char __user *__user *argv,
char __user *__user *envp,
struct pt_regs * regs)
{
struct linux_binprm *bprm;
struct file *file;
int retval;
int i;

retval = -ENOMEM;
bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
if (!bprm)
    goto out_ret;

file = open_exec(filename);
retval = PTR_ERR(file);
if (IS_ERR(file))
    goto out_kfree;

sched_exec();

bprm->p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);

bprm->file = file;
bprm->filename = filename;
bprm->interp = filename;
bprm->mm = mm_alloc();
retval = -ENOMEM;
if (!bprm->mm)
    goto out_file;

retval = init_new_context(current, bprm->mm);
if (retval < 0)
    goto out_mm;

bprm->argc = count(argv, bprm->p / sizeof(void *));
```

```
    if ((retval = bprm->argc) < 0)
        goto out_mm;

    bprm->envc = count(envp, bprm->p / sizeof(void *));
    if ((retval = bprm->envc) < 0)
        goto out_mm;

    retval = security_bprm_alloc(bprm);
    if (retval)
        goto out;

    retval = prepare_binprm(bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    if (retval < 0)
        goto out;

    bprm->exec = bprm->p;
    retval = copy_strings(bprm->envc, envp, bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings(bprm->argc, argv, bprm);
    if (retval < 0)
        goto out;

    retval = search_binary_handler(bprm,regs);
    if (retval >= 0) {
        free_arg_pages(bprm);

        /* execve success */
        security_bprm_free(bprm);
        acct_update_integrals(current);
        kfree(bprm);
        return retval;
    }

    out:
    /* Something went wrong, return the inode and free the argument pages*/
    for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
        struct page * page = bprm->page[i];
        if (page)
```

```
        __free_page(page);
    }

    if (bprm->security)
        security_bprm_free(bprm);

    out_mm:
    if (bprm->mm)
        mmdrop(bprm->mm);

    out_file:
    if (bprm->file) {
        allow_write_access(bprm->file);
        fput(bprm->file);
    }

    out_kfree:
    kfree(bprm);

    out_ret:
    return retval;
    }
```

do_execve分析

```
/*
 * sys_execve() executes a new program.
 */
int do_execve(char * filename,
    char __user *__user *argv,
    char __user *__user *envp,
    struct pt_regs * regs)
{
```

这里的 filename 已经位于内核空间，但启动该可执行文件的参数列表 argv[]还在用户空间，同样该程序执行的环境列表 envp[]同样也在用户空间。

```
    struct linux_binprm *bprm;
    struct file *file;
    int retval;
    int i;

    retval = -ENOMEM;
    bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
    if (!bprm)
        goto out_ret;
```

kzalloc()的注释如下:

```
/**
 * kzalloc - allocate memory. The memory is set to zero.
 * @size: how many bytes of memory are required.
 * @flags: the type of memory to allocate (see kmalloc).
 */

    file = open_exec(filename);
    retval = PTR_ERR(file);
    if (IS_ERR(file))
        goto out_kfree;
```

打开可执行文件。该函数名曰"open_exec",实际上是一个在内核通用的打开文件的函数，且其 symbol 也被内核输出，

```
EXPORT_SYMBOL(open_exec);
```

故 LKM 的开发者完全可以调用此函数。另外提一下，有了 open，自然应该有 read, write, close 的函数。

read()

```
int kernel_read(struct file *file, unsigned long offset,    char
*addr, unsigned long count)

close()
void fastcall __fput(struct file *file)
```

那么 write 与 create 文件呢？（不需要 seek,应为读写都带有文件位置指针）。在内核态的文件操作不象在用户态那么方便，值得写篇文章讨论讨论。在网上搜了一把，有人提问，有人给出似是而非的答案。没看到满意的。

```
    sched_exec();
```

上面代码的作者对该函数的注释如下:

```
/*
 * sched_exec - execve() is a valuable balancing opportunity, because at
 * this point the task has the smallest effective memory and cache
footprint.
 */
```

该函数显然与 schedule 有关。 2.6 内核的 schedule 被彻底重写，还没读过呢。该函数应该与本执行文件的载入关系不大。

```
    bprm->p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
```

bprm->p 指向分配给该 process 的 argv，envp 空间的顶部。实际上就是未来该 process 的用户 stack 的栈顶（现在当然还不能这么说，因为该空间还没有被映射到虚拟地址空间中）argv，envp 空间大小为 32 * 4K = 128K,而该栈顶为 128K - 4 = 0x1FFFC。

```
    bprm->file = file;
```

纪录该可执行文件的打开的 file structure。

```
    bprm->filename = filename;
```

纪录该可执行文件的文件名。

```
    bprm->interp = filename;
```

先假设该可执行文件可能用到的动态链接器的文件名与可执行文件相同。这个假设几乎 100%不成立。

```
    bprm->mm = mm_alloc();
    retval = -ENOMEM;
    if (!bprm->mm)
        goto out_file;
```

分配进程的内存管理的结构。

```
    retval = init_new_context(current, bprm->mm);
    if (retval < 0)
        goto out_mm;
```

初始化刚分配的该进程的内存管理结构。

```
    bprm->argc = count(argv, bprm->p / sizeof(void *));
    if ((retval = bprm->argc) < 0)
        goto out_mm;
```

这里 argc 即是 argv[]数组的个数，同 int main(int argc, char* argv[])中的 argc 一样。只不过由于 argv[]在用户空间，而计数的动作在内核空间，计起来有点麻烦而已。比如:
$ /bin/ls -l -a
这里 argc = 3，argv[0] = /bin/ls， argv[1] = -l， argv[2] = -a

```
    bprm->envc = count(envp, bprm->p / sizeof(void *));
    if ((retval = bprm->envc) < 0)
        goto out_mm;
```

同样 bprm->envc 计数当前 process 的环境变量数组的个数。学过 C 语言的 programmer 不知还记得吗，main()函数有多种形式，其中最完整的签名如下:

int main(int argc, char *argv[],char *env[]);

这里的 env[]即是环境变量数组。类似于你在 shell 里输入如下命令后看到的。

```
[wzhou@dcmp10 ~]$ env
HOSTNAME=dcmp10
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=::ffff:13.187.241.182 1651 22
SSH_TTY=/dev/pts/9
USER=wzhou
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01
:cd=40;33;0
1:or=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe=00;32:*.co
m=00;32:*.b
tm=00;32:*.bat=00;32:*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*
.arj=00;31:
*.taz=00;31:*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.
bz2=00;31:*
       .bz=00;31:*.tz=00;31:*.rpm=00;31:*.cpio=00;31:*.jpg=00;35:*.gi
f=00;35:*.bmp=00;3
5:*.xbm=00;35:*.xpm=00;35:*.png=00;35:*.tif=00;35:
KDEDIR=/usr
MAIL=/var/spool/mail/wzhou
PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/h
ome/wzhou/a
ld-0.1.7/bin:/home/wzhou/bin
```

```
INPUTRC=/etc/inputrc
PWD=/home/wzhou
LANG=en_US.UTF-8
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SHLVL=1
HOME=/home/wzhou
LOGNAME=wzhou
SSH_CONNECTION=::ffff:13.187.241.182 1651 ::ffff:13.187.243.58 22
LESSOPEN=|/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
_=/bin/env
```

对，东西确实挺多的。

```
    retval = security_bprm_alloc(bprm);
    if (retval)
        goto out;
```

看函数名与安全有关。只有在内核编译选项 CONFIG_SECURITY 打开的情况下，才会有意义，否则就为空。

```
static inline int security_bprm_alloc (struct linux_binprm *bprm)
{
    return 0;
}
```

忽略它，不影响我们理解。

```
    retval = prepare_binprm(bprm);
    if (retval < 0)
        goto out;
```

见 prepare_binprm()的分析。

```
    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    if (retval < 0)
        goto out;
```

拷贝可执行文件名。

```
    bprm->exec = bprm->p;
    retval = copy_strings(bprm->envc, envp, bprm);
    if (retval < 0)
        goto out;
```

复制当前 envp[]的内容（注意，当前 process 的地址空间还是继承自父进程的，完全与父进程的相同，自然这里的 aregv[]的内存中的内容也与父进程的相同。）这里特别说明一下，常规运行一个程序的步骤是这样的（当然从 programmer 的角度）:

1. 先 fork()一个子进程，这个子进程的地址空间是与父进程相同的，自然父进程的 argv[]

与 envp[]也被子进程继承。

2. 子进程会调用 execve()系统调用，以运行新程序。在 execve()中它当然会先废弃掉原来的地址空间（可以理解吗，因为新执行的程序与父进程完全是两个不同的程序吗，自然有完全不同的地址分配吗），然后建立新的地址空间。我们先在正在分析 execve()调用，目前还没到废弃父进程的地址空间的时候，所以现在整个地址空间还与父进程的相同。

```
    retval = copy_strings(bprm->argc, argv, bprm);
    if (retval < 0)
        goto out;
```

复制当前 argv[]中的内容。具体请看对 copy_strings（）的分析。

```
    retval = search_binary_handler(bprm,regs);
```

所有的可执行文件的处理器都被链接在全局变量 formats 定义的链表里面。

在 src/fs/exec.c 中的

```
    static struct linux_binfmt *formats;
```

函数 int register_binfmt(struct linux_binfmt * fmt)完成向该链表添加节点，而 int unregister_binfmt(struct linux_binfmt * fmt)则删除节点。代码非常简单，就不介绍了。

search_binary_handler()函数在枚举该链表，依次让该链表上的处理器来辨认，具体见该函数的分析。

```
    if (retval >= 0) {
```

如果找不到对应的 executable loader 来载入该可执行文件，则释放掉申请的资源，然后返回。

```
        free_arg_pages(bprm);
```

释放那存放 argv[]与 envp[]的最多 32 个页框（在 copy_strings 函数里分配）。

```
        /* execve success */
        security_bprm_free(bprm);
```

现在是个空函数。

```
static inline void security_bprm_free (struct linux_binprm *bprm)
{ }
```

```
        acct_update_integrals(current);
```

更新该 process 与内存相关的 accounting 值。

```
        kfree(bprm);
```

释放在本函数开始时申请的空间。现在用户空间是全新的，不在是父进程的，但在内核空间却还得好好料理。

```
        return retval;
    }
```

```
out:
    /* Something went wrong, return the inode and free the argument pages*/
    for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
        struct page * page = bprm->page[i];
        if (page)
            __free_page(page);
    }
```

bprm->page[i]在 copy_strings()中申请，具体请见该函数的注释。

```
    if (bprm->security)
        security_bprm_free(bprm);
```

与 security 相关，忽略。

下面都是如果在某一步失败，对应的资源要释放。

```
out_mm:
    if (bprm->mm)
        mmdrop(bprm->mm);

out_file:
    if (bprm->file) {
        allow_write_access(bprm->file);
        fput(bprm->file);
    }

out_kfree:
    kfree(bprm);

out_ret:
    return retval;
}
```

prepare_binprm源码

```
/*
 * Fill the binprm structure from the inode.
 * Check permissions, then read the first 128 (BINPRM_BUF_SIZE) bytes
 */
int prepare_binprm(struct linux_binprm *bprm)
{
int mode;
struct inode * inode = bprm->file->f_path.dentry->d_inode;
int retval;

mode = inode->i_mode;
if (bprm->file->f_op == NULL)
    return -EACCES;

bprm->e_uid = current->euid;
bprm->e_gid = current->egid;

if(!(bprm->file->f_path.mnt->mnt_flags & MNT_NOSUID)) {
    /* Set-uid? */
    if (mode & S_ISUID) {
        current->personality &= ~PER_CLEAR_ON_SETID;
        bprm->e_uid = inode->i_uid;
    }

    /* Set-gid? */
    /*
     * If setgid is set but no group execute bit then this
     * is a candidate for mandatory locking, not a setgid
     * executable.
     */
    if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP)) {
        current->personality &= ~PER_CLEAR_ON_SETID;
        bprm->e_gid = inode->i_gid;
    }
}

/* fill in binprm security blob */
retval = security_bprm_set(bprm);
if (retval)
    return retval;
```

```
    memset(bprm->buf,0,BINPRM_BUF_SIZE);
    return kernel_read(bprm->file,0,bprm->buf,BINPRM_BUF_SIZE);
    }
```

prepare_binprm注释

```
/*
 * Fill the binprm structure from the inode.
 * Check permissions, then read the first 128 (BINPRM_BUF_SIZE) bytes
 */
int prepare_binprm(struct linux_binprm *bprm)
{
    int mode;
    struct inode * inode = bprm->file->f_path.dentry->d_inode;
    int retval;

    mode = inode->i_mode;
    if (bprm->file->f_op == NULL)
        return -EACCES;

    bprm->e_uid = current->euid;
    bprm->e_gid = current->egid;
```

设置有效用户 id 与有效组 id。至于什么叫"有效用户 id 与有效组 id"?与现在分析的 ELF 可执行文件的载入不怎么搭界。

```
    if(!(bprm->file->f_path.mnt->mnt_flags & MNT_NOSUID)) {
```

这里是针对 mount 时如果没有加 MNT_NOSUID 选项的处理。
mount 手册中对该选项的描述如下:

```
nosuid
Do  not  allow set-user-identifier or set-group-identifier bits to take
effect. (This seems safe, but is in fact rather unsafe if you have
suidperl(1) installed.)
```

所谓 suid 是指可执行文件在执行时用该文件所有者的 id 而非当前用户的 id。比如 passwd 就是这样一条命令。具体有什么好处，啰哩啰嗦一大堆，请参看任何一本介绍 Unix 的书。

```
        /* Set-uid? */
        if (mode & S_ISUID) {
            current->personality &= ~PER_CLEAR_ON_SETID;
            bprm->e_uid = inode->i_uid;
        }
```

如果该可执行文件设了 SUID 标志，把有效用户 id 设成该文件拥有者的用户 id，也就是该程序有了可执行文件拥有者的权利。比如我是普通用户，但该程序的拥有者是 root，那这个程序拥有同 root 同样的权利。这也是所谓 effective user id 中 “effective” 的意思。

```
        /* Set-gid? */
        /*
         * If setgid is set but no group execute bit then this
         * is a candidate for mandatory locking, not a setgid
         * executable.
         */
        if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP)) {
            current->personality &= ~PER_CLEAR_ON_SETID;
            bprm->e_gid = inode->i_gid;
```

如果该可执行文件设了 SGID 标志，把有效用户 id 设成该文件拥有者的组 id。使得该程序拥有文件拥有组一样的权利。

```
        }
    }

    /* fill in binprm security blob */
    retval = security_bprm_set(bprm);
    if (retval)
        return retval;
```

与安全相关，忽略。没办法，我也没研究过。

```
    memset(bprm->buf,0,BINPRM_BUF_SIZE);
    return kernel_read(bprm->file,0,bprm->buf,BINPRM_BUF_SIZE);
```

读取该可执行文件的头上 128 个 byte，因为一般可执行文件其头上包含该 format 的 header，比如 aout header，ELF header 等。从这 128 个 byte 就能判断出是否是该 executable loader 关心的。

```
}
```

copy_strings源码

```
/*
 * 'copy_strings()' copies argument/environment strings from user
 * memory to free pages in kernel mem. These are in a format ready
 * to be put directly into the top of new user memory.
 */
static int copy_strings(int argc, char __user * __user * argv,
        struct linux_binprm *bprm)
{
struct page *kmapped_page = NULL;
char *kaddr = NULL;
int ret;

while (argc-- > 0) {
    char __user *str;
    int len;
    unsigned long pos;

    if (get_user(str, argv+argc) ||
            !(len = strnlen_user(str, bprm->p))) {
        ret = -EFAULT;
        goto out;
    }

    if (bprm->p < len)  {
        ret = -E2BIG;
        goto out;
    }

    bprm->p -= len;
    /* XXX: add architecture specific overflow check here. */
    pos = bprm->p;

    while (len > 0) {
        int i, new, err;
        int offset, bytes_to_copy;
        struct page *page;

        offset = pos % PAGE_SIZE;
        i = pos/PAGE_SIZE;
        page = bprm->page[i];
```

```
            new = 0;
            if (!page) {
                page = alloc_page(GFP_HIGHUSER);
                bprm->page[i] = page;
                if (!page) {
                    ret = -ENOMEM;
                    goto out;
                }
                new = 1;
            }

            if (page != kmapped_page) {
                if (kmapped_page)
                    kunmap(kmapped_page);
                kmapped_page = page;
                kaddr = kmap(kmapped_page);
            }
            if (new && offset)
                memset(kaddr, 0, offset);
            bytes_to_copy = PAGE_SIZE - offset;
            if (bytes_to_copy > len) {
                bytes_to_copy = len;
                if (new)
                    memset(kaddr+offset+len, 0,
                        PAGE_SIZE-offset-len);
            }
            err = copy_from_user(kaddr+offset, str, bytes_to_copy);
            if (err) {
                ret = -EFAULT;
                goto out;
            }

            pos += bytes_to_copy;
            str += bytes_to_copy;
            len -= bytes_to_copy;
        }
    }
    ret = 0;
    out:
    if (kmapped_page)
        kunmap(kmapped_page);
    return ret;
    }
```

copy_strings注释

本来只是复制字符串，功能类似于标准 C 库的 strcpy()函数，但这里有两大看点

1. 从用户空间复制字符串到内核空间
2. 这复制的字符串不是普通的字符串，而是对任何 C 程序员都有特殊意义的。C programmer 都耳熟能详 main()函数。

int main(int argc, char *argv[], char* envp[]);

main()函数是所有 C/C++程序的开始,那传递给 main()的参数是哪儿来的呢？这里就是答案。这里 Linux 的实现与 Windows 的就不一样。Linux 是实现在内核里面的，而 Windows 是实现在 C 库里面的。最新的 Visual Studio 包含了 Windows 版 C 库的源代码，可以参考。

```
/*
 * 'copy_strings()' copies argument/environment strings from user
 * memory to free pages in kernel mem. These are in a format ready
 * to be put directly into the top of new user memory.
 */
static int copy_strings(int argc, char __user * __user * argv,
          struct linux_binprm *bprm)
{
```

这里*argv[]是在用户态里的字符串数组，而 argc 是该数组的长度。bprm 参数结构里面的 page[]空间用于存放来自用户态的字符串数组。

```
    struct page *kmapped_page = NULL;
    char *kaddr = NULL;
    int ret;

    while (argc-- > 0) {
```

该循环用于枚举 argvp[]中的字符串。即这里 argc 是 argv[]数组的个数。

比如:

```
argv[0] = HOSTNAME=dcmp10
argv[1] = TERM=xterm
argv[2] = SHELL=/bin/bash
argv[3] = HISTSIZE=1000
argv[4] = SSH_CLIENT=::ffff:13.187.241.182 1986 22
argv[5] = SSH_TTY=/dev/pts/2
argv[6] = USER=wzhou
…

argv[28] = LOGNAME=wzhou
argv[29] = _=/bin/env
```

上例中 argc = 30。这里的循环就是对该数组的枚举，但有一点切记，这个数组的空间在用户

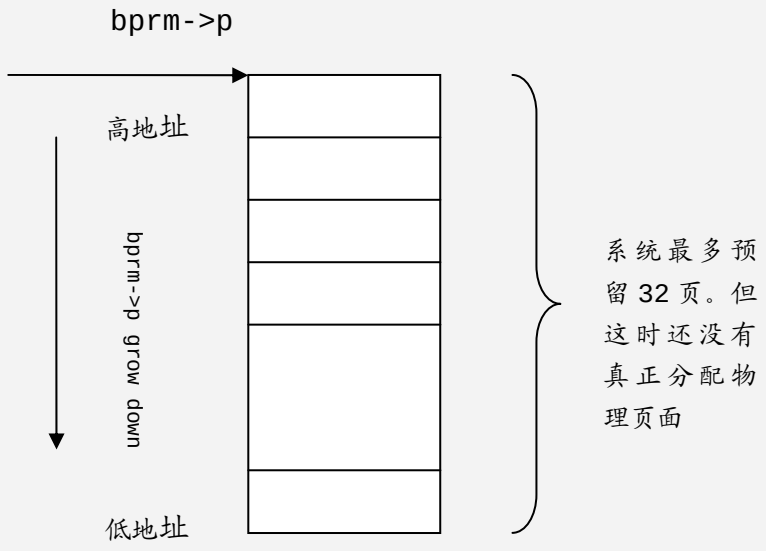
空间，否则也不用这么麻烦了。

```
        char __user *str;
        int len;
        unsigned long pos;

        if (get_user(str, argv+argc) ||
                !(len = strnlen_user(str, bprm->p))) {
            ret = -EFAULT;
            goto out;
        }
```

类似于 C 库中的 strlen()计算 string 长度，还是由于 string 在用户空间才这么麻烦。返回的 len 包括最后的 NULL 字符，这与 strcpy()不同。如果用户态的空间有问题，会被 get_user()和 strnlen_user()捕捉到，返回失败。

```
        if (bprm->p < len)  {
            ret = -E2BIG;
            goto out;
        }
```

存放用户态的 argv[]与 envp[]的内核空间不够，表示用户态的参数列表与环境变量列表变态的长，总长超过 32 个叶面，128K，报错。



正象上图所绘的，bprm->p 指针是由高地址向低地址生长的，最初始在顶端(128K - 4)。在 do_execve()中有下面的赋值

```
    bprm->p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
```

这里只是 128K 空间的 offset，还谈不上是地址，因为该物理空间还未被 mapping 入虚拟地址空间。

```
        bprm->p -= len;
```
检查是否超出范围。

```
        /* XXX: add architecture specific overflow check here. */
        pos = bprm->p;

        while (len > 0) {
            int i, new, err;
            int offset, bytes_to_copy;
            struct page *page;

            offset = pos % PAGE_SIZE;
            i = pos/PAGE_SIZE;
```
i 用于确定当前要拷贝的位置在 128K，32 页中位于哪一页；而 offset 则是该页的偏移。为什么这么麻烦呢，关键是这些内存还没有被映射进地址空间，根本没有“virtual address”的概念。就象大街上的某座房子，还没有给编上门牌号码，你自然无法通过什么大街第几号来找到它。

```
            page = bprm->page[i];
```
这里的 page 管理的是物理页面，不是可寻址的虚拟空间。

```
            new = 0;
            if (!page) {
                page = alloc_page(GFP_HIGHUSER);
```
如果该物理页面还没有分配，则通过物理页面分配器申请一页。这里的物理页面分配器是系统中所有内存分配之母，是最底层的内存管理器。一切内存管理都建立在其之上。

```
                bprm->page[i] = page;
                if (!page) {
                    ret = -ENOMEM;
                    goto out;
                }
                new = 1;
```
设置分配物理页面标志。以便下面判断是否要初始化。
```
            }

            if (page != kmapped_page) {
                if (kmapped_page)
                    kunmap(kmapped_page);
                kmapped_page = page;
                kaddr = kmap(kmapped_page);
```
正象上面说的，page 只是物理页面，是不能寻址的，所以这里临时把它映射到 Virtual Address 中，就象给没有门牌号码的房子先指定一个号码。

```
            }
            if (new && offset)
                memset(kaddr, 0, offset);
            bytes_to_copy = PAGE_SIZE - offset;
            if (bytes_to_copy > len) {
                bytes_to_copy = len;
                if (new)
                    memset(kaddr+offset+len, 0,
                        PAGE_SIZE-offset-len);
            }
```

清零初始化。

```
            err = copy_from_user(kaddr+offset, str, bytes_to_copy);
            if (err) {
                ret = -EFAULT;
                goto out;
            }
```

这儿才是本函数实质性的内容，把 argv[i]的字符串从用户态拷贝过来。这里 kaddr 就是临时“门牌号码”。

```
            pos += bytes_to_copy;
            str += bytes_to_copy;
            len -= bytes_to_copy;
```

调整指针，为拷贝 argv[i + 1]做准备。

```
        }
    }
    ret = 0;
out:
    if (kmapped_page)
        kunmap(kmapped_page);
```

取消临时“门牌号码”。

```
    return ret;
}
```

load_elf_binary源码

```
static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs
*regs)
   {
   struct file *interpreter = NULL; /* to shut gcc up */
       unsigned long load_addr = 0, load_bias = 0;
   int load_addr_set = 0;
   char * elf_interpreter = NULL;
   unsigned int interpreter_type = INTERPRETER_NONE;
   unsigned char ibcs2_interpreter = 0;
   unsigned long error;
   struct elf_phdr *elf_ppnt, *elf_phdata;
   unsigned long elf_bss, elf_brk;
   int elf_exec_fileno;
   int retval, i;
   unsigned int size;
   unsigned long elf_entry, interp_load_addr = 0;
   unsigned long start_code, end_code, start_data, end_data;
   unsigned long reloc_func_desc = 0;
   char passed_fileno[6];
   struct files_struct *files;
   int executable_stack = EXSTACK_DEFAULT;
   unsigned long def_flags = 0;
   struct {
       struct elfhdr elf_ex;
       struct elfhdr interp_elf_ex;
           struct exec interp_ex;
   } *loc;

   loc = kmalloc(sizeof(*loc), GFP_KERNEL);
   if (!loc) {
       retval = -ENOMEM;
       goto out_ret;
   }

   /* Get the exec-header */
   loc->elf_ex = *((struct elfhdr *)bprm->buf);

   retval = -ENOEXEC;
   /* First of all, some simple consistency checks */
   if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
       goto out;
```

```
    if (loc->elf_ex.e_type != ET_EXEC && loc->elf_ex.e_type != ET_DYN)
        goto out;
    if (!elf_check_arch(&loc->elf_ex))
        goto out;
    if (!bprm->file->f_op||!bprm->file->f_op->mmap)
        goto out;

    /* Now read in all of the header information */
    if (loc->elf_ex.e_phentsize != sizeof(struct elf_phdr))
        goto out;
    if (loc->elf_ex.e_phnum < 1 ||
        loc->elf_ex.e_phnum > 65536U / sizeof(struct elf_phdr))
        goto out;
    size = loc->elf_ex.e_phnum * sizeof(struct elf_phdr);
    retval = -ENOMEM;
    elf_phdata = kmalloc(size, GFP_KERNEL);
    if (!elf_phdata)
        goto out;

    retval = kernel_read(bprm->file, loc->elf_ex.e_phoff,
                (char *)elf_phdata, size);
    if (retval != size) {
        if (retval >= 0)
            retval = -EIO;
        goto out_free_ph;
    }

    files = current->files;  /* Refcounted so ok */
    retval = unshare_files();
    if (retval < 0)
        goto out_free_ph;
    if (files == current->files) {
        put_files_struct(files);
        files = NULL;
    }

    /* exec will make our files private anyway, but for the a.out
       loader stuff we need to do it earlier */
    retval = get_unused_fd();
    if (retval < 0)
        goto out_free_fh;
    get_file(bprm->file);
    fd_install(elf_exec_fileno = retval, bprm->file);
```

```
elf_ppnt = elf_phdata;
elf_bss = 0;
elf_brk = 0;

start_code = ~0UL;
end_code = 0;
start_data = 0;
end_data = 0;

for (i = 0; i < loc->elf_ex.e_phnum; i++) {
    if (elf_ppnt->p_type == PT_INTERP) {
        /* This is the program interpreter used for
         * shared libraries - for now assume that this
         * is an a.out format binary
         */
        retval = -ENOEXEC;
        if (elf_ppnt->p_filesz > PATH_MAX ||
            elf_ppnt->p_filesz < 2)
            goto out_free_file;

        retval = -ENOMEM;
        elf_interpreter = kmalloc(elf_ppnt->p_filesz,
                    GFP_KERNEL);
        if (!elf_interpreter)
            goto out_free_file;

        retval = kernel_read(bprm->file, elf_ppnt->p_offset,
                   elf_interpreter,
                   elf_ppnt->p_filesz);
        if (retval != elf_ppnt->p_filesz) {
            if (retval >= 0)
                retval = -EIO;
            goto out_free_interp;
        }
        /* make sure path is NULL terminated */
        retval = -ENOEXEC;
        if (elf_interpreter[elf_ppnt->p_filesz - 1] != '\0')
            goto out_free_interp;

        /* If the program interpreter is one of these two,
         * then assume an iBCS2 image. Otherwise assume
         * a native linux image.
         */
```

```
        if (strcmp(elf_interpreter,"/usr/lib/libc.so.1") == 0 ||
            strcmp(elf_interpreter,"/usr/lib/ld.so.1") == 0)
            ibcs2_interpreter = 1;

        /*
         * The early SET_PERSONALITY here is so that the lookup
         * for the interpreter happens in the namespace of the
         * to-be-execed image.  SET_PERSONALITY can select an
         * alternate root.
         *
         * However, SET_PERSONALITY is NOT allowed to switch
         * this task into the new images's memory mapping
         * policy - that is, TASK_SIZE must still evaluate to
         * that which is appropriate to the execing application.
         * This is because exit_mmap() needs to have TASK_SIZE
         * evaluate to the size of the old image.
         *
         * So if (say) a 64-bit application is execing a 32-bit
         * application it is the architecture's responsibility
         * to defer changing the value of TASK_SIZE until the
         * switch really is going to happen - do this in
         * flush_thread().    - akpm
         */
        SET_PERSONALITY(loc->elf_ex, ibcs2_interpreter);

        interpreter = open_exec(elf_interpreter);
        retval = PTR_ERR(interpreter);
        if (IS_ERR(interpreter))
            goto out_free_interp;

        /*
         * If the binary is not readable then enforce
         * mm->dumpable = 0 regardless of the interpreter's
         * permissions.
         */
        if (file_permission(interpreter, MAY_READ) < 0)
            bprm->interp_flags |= BINPRM_FLAGS_ENFORCE_NONDUMP;

        retval = kernel_read(interpreter, 0, bprm->buf,
                  BINPRM_BUF_SIZE);
        if (retval != BINPRM_BUF_SIZE) {
            if (retval >= 0)
                retval = -EIO;
            goto out_free_dentry;
```

```
            }

            /* Get the exec headers */
            loc->interp_ex = *((struct exec *)bprm->buf);
            loc->interp_elf_ex = *((struct elfhdr *)bprm->buf);
            break;
        }
        elf_ppnt++;
    }

    elf_ppnt = elf_phdata;
    for (i = 0; i < loc->elf_ex.e_phnum; i++, elf_ppnt++)
        if (elf_ppnt->p_type == PT_GNU_STACK) {
            if (elf_ppnt->p_flags & PF_X)
                executable_stack = EXSTACK_ENABLE_X;
            else
                executable_stack = EXSTACK_DISABLE_X;
            break;
        }

    /* Some simple consistency checks for the interpreter */
    if (elf_interpreter) {
        interpreter_type = INTERPRETER_ELF | INTERPRETER_AOUT;

        /* Now figure out which format our binary is */
        if ((N_MAGIC(loc->interp_ex) != OMAGIC) &&
            (N_MAGIC(loc->interp_ex) != ZMAGIC) &&
            (N_MAGIC(loc->interp_ex) != QMAGIC))
            interpreter_type = INTERPRETER_ELF;

        if (memcmp(loc->interp_elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
            interpreter_type &= ~INTERPRETER_ELF;

        retval = -ELIBBAD;
        if (!interpreter_type)
            goto out_free_dentry;

        /* Make sure only one type was selected */
        if ((interpreter_type & INTERPRETER_ELF) &&
             interpreter_type != INTERPRETER_ELF) {
                // FIXME - ratelimit this before re-enabling
            // printk(KERN_WARNING "ELF: Ambiguous type, using ELF\n");
            interpreter_type = INTERPRETER_ELF;
        }
```

```
        /* Verify the interpreter has a valid arch */
        if ((interpreter_type == INTERPRETER_ELF) &&
            !elf_check_arch(&loc->interp_elf_ex))
            goto out_free_dentry;
    } else {
        /* Executables without an interpreter also need a personality  */
        SET_PERSONALITY(loc->elf_ex, ibcs2_interpreter);
    }

    /* OK, we are done with that, now set up the arg stuff,
       and then start this sucker up */
    if ((!bprm->sh_bang) && (interpreter_type == INTERPRETER_AOUT)) {
        char *passed_p = passed_fileno;
        sprintf(passed_fileno, "%d", elf_exec_fileno);

        if (elf_interpreter) {
            retval = copy_strings_kernel(1, &passed_p, bprm);
            if (retval)
                goto out_free_dentry;
            bprm->argc++;
        }
    }

    /* Flush all traces of the currently running executable */
    retval = flush_old_exec(bprm);
    if (retval)
        goto out_free_dentry;

    /* Discard our unneeded old files struct */
    if (files) {
        put_files_struct(files);
        files = NULL;
    }

    /* OK, This is the point of no return */
    current->mm->start_data = 0;
    current->mm->end_data = 0;
    current->mm->end_code = 0;
    current->mm->mmap = NULL;
    current->flags &= ~PF_FORKNOEXEC;
    current->mm->def_flags = def_flags;

    /* Do this immediately, since STACK_TOP as used in setup_arg_pages
       may depend on the personality.  */
```

```
    SET_PERSONALITY(loc->elf_ex, ibcs2_interpreter);
    if (elf_read_implies_exec(loc->elf_ex, executable_stack))
        current->personality |= READ_IMPLIES_EXEC;

    if     (!(current->personality     &     ADDR_NO_RANDOMIZE)     &&
randomize_va_space)
        current->flags |= PF_RANDOMIZE;
    arch_pick_mmap_layout(current->mm);

    /* Do this so that we can load the interpreter, if need be.  We will
       change some of these later */
    current->mm->free_area_cache = current->mm->mmap_base;
    current->mm->cached_hole_size = 0;
    retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                executable_stack);
    if (retval < 0) {
        send_sig(SIGKILL, current, 0);
        goto out_free_dentry;
    }

    current->mm->start_stack = bprm->p;

    /* Now we do a little grungy work by mmaping the ELF image into
       the correct location in memory.  At this point, we assume that
       the image should be loaded at fixed address, not at a variable
       address. */
    for(i = 0, elf_ppnt = elf_phdata;
        i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
        int elf_prot = 0, elf_flags;
        unsigned long k, vaddr;

        if (elf_ppnt->p_type != PT_LOAD)
            continue;

        if (unlikely (elf_brk > elf_bss)) {
            unsigned long nbyte;

            /* There was a PT_LOAD segment with p_memsz > p_filesz
               before this one. Map anonymous pages, if needed,
               and clear the area.  */
            retval = set_brk (elf_bss + load_bias,
                      elf_brk + load_bias);
            if (retval) {
                send_sig(SIGKILL, current, 0);
```

```
            goto out_free_dentry;
        }
        nbyte = ELF_PAGEOFFSET(elf_bss);
        if (nbyte) {
            nbyte = ELF_MIN_ALIGN - nbyte;
            if (nbyte > elf_brk - elf_bss)
                nbyte = elf_brk - elf_bss;
            if (clear_user((void __user *)elf_bss +
                    load_bias, nbyte)) {
                /*
                 * This bss-zeroing can fail if the ELF
                 * file specifies odd protections. So
                 * we don't check the return value
                 */
            }
        }
    }

    if (elf_ppnt->p_flags & PF_R)
        elf_prot |= PROT_READ;
    if (elf_ppnt->p_flags & PF_W)
        elf_prot |= PROT_WRITE;
    if (elf_ppnt->p_flags & PF_X)
        elf_prot |= PROT_EXEC;

    elf_flags = MAP_PRIVATE | MAP_DENYWRITE | MAP_EXECUTABLE;

    vaddr = elf_ppnt->p_vaddr;
    if (loc->elf_ex.e_type == ET_EXEC || load_addr_set) {
        elf_flags |= MAP_FIXED;
    } else if (loc->elf_ex.e_type == ET_DYN) {
        /* Try and get dynamic programs out of the way of the
         * default mmap base, as well as whatever program they
         * might try to exec.  This is because the brk will
         * follow the loader, and is not movable.  */
        load_bias = ELF_PAGESTART(ELF_ET_DYN_BASE - vaddr);
    }

    error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt,
            elf_prot, elf_flags);
    if (BAD_ADDR(error)) {
        send_sig(SIGKILL, current, 0);
        goto out_free_dentry;
    }
```

```
        if (!load_addr_set) {
            load_addr_set = 1;
            load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
            if (loc->elf_ex.e_type == ET_DYN) {
                load_bias += error -
                             ELF_PAGESTART(load_bias + vaddr);
                load_addr += load_bias;
                reloc_func_desc = load_bias;
            }
        }
        k = elf_ppnt->p_vaddr;
        if (k < start_code)
            start_code = k;
        if (start_data < k)
            start_data = k;

        /*
         * Check to see if the section's size will overflow the
         * allowed task size. Note that p_filesz must always be
         * <= p_memsz so it is only necessary to check p_memsz.
         */
        if (BAD_ADDR(k) || elf_ppnt->p_filesz > elf_ppnt->p_memsz ||
            elf_ppnt->p_memsz > TASK_SIZE ||
            TASK_SIZE - elf_ppnt->p_memsz < k) {
            /* set_brk can never work. Avoid overflows. */
            send_sig(SIGKILL, current, 0);
            goto out_free_dentry;
        }

        k = elf_ppnt->p_vaddr + elf_ppnt->p_filesz;

        if (k > elf_bss)
            elf_bss = k;
        if ((elf_ppnt->p_flags & PF_X) && end_code < k)
            end_code = k;
        if (end_data < k)
            end_data = k;
        k = elf_ppnt->p_vaddr + elf_ppnt->p_memsz;
        if (k > elf_brk)
            elf_brk = k;
    }

    loc->elf_ex.e_entry += load_bias;
```

```
    elf_bss += load_bias;
    elf_brk += load_bias;
    start_code += load_bias;
    end_code += load_bias;
    start_data += load_bias;
    end_data += load_bias;

    /* Calling set_brk effectively mmaps the pages that we need
     * for the bss and break sections.  We must do this before
     * mapping in the interpreter, to make sure it doesn't wind
     * up getting placed where the bss needs to go.
     */
    retval = set_brk(elf_bss, elf_brk);
    if (retval) {
        send_sig(SIGKILL, current, 0);
        goto out_free_dentry;
    }
    if (likely(elf_bss != elf_brk) && unlikely(padzero(elf_bss))) {
        send_sig(SIGSEGV, current, 0);
        retval = -EFAULT; /* Nobody gets to see this, but.. */
        goto out_free_dentry;
    }

    if (elf_interpreter) {
        if (interpreter_type == INTERPRETER_AOUT)
            elf_entry = load_aout_interp(&loc->interp_ex,
                                         interpreter);
        else
            elf_entry = load_elf_interp(&loc->interp_elf_ex,
                                        interpreter,
                                        &interp_load_addr);
        if (BAD_ADDR(elf_entry)) {
            force_sig(SIGSEGV, current);
            retval = IS_ERR((void *)elf_entry) ?
                    (int)elf_entry : -EINVAL;
            goto out_free_dentry;
        }
        reloc_func_desc = interp_load_addr;

        allow_write_access(interpreter);
        fput(interpreter);
        kfree(elf_interpreter);
    } else {
        elf_entry = loc->elf_ex.e_entry;
```

```
        if (BAD_ADDR(elf_entry)) {
            force_sig(SIGSEGV, current);
            retval = -EINVAL;
            goto out_free_dentry;
        }
    }

    kfree(elf_phdata);

    if (interpreter_type != INTERPRETER_AOUT)
        sys_close(elf_exec_fileno);

    set_binfmt(&elf_format);

    #ifdef ARCH_HAS_SETUP_ADDITIONAL_PAGES
    retval = arch_setup_additional_pages(bprm, executable_stack);
    if (retval < 0) {
        send_sig(SIGKILL, current, 0);
        goto out;
    }
    #endif /* ARCH_HAS_SETUP_ADDITIONAL_PAGES */

    compute_creds(bprm);
    current->flags &= ~PF_FORKNOEXEC;
    create_elf_tables(bprm, &loc->elf_ex,
            (interpreter_type == INTERPRETER_AOUT),
            load_addr, interp_load_addr);
    /* N.B. passed_fileno might not be initialized? */
    if (interpreter_type == INTERPRETER_AOUT)
        current->mm->arg_start += strlen(passed_fileno) + 1;
    current->mm->end_code = end_code;
    current->mm->start_code = start_code;
    current->mm->start_data = start_data;
    current->mm->end_data = end_data;
    current->mm->start_stack = bprm->p;

    if (current->personality & MMAP_PAGE_ZERO) {
        /* Why this, you ask???  Well SVr4 maps page 0 as read-only,
           and some applications "depend" upon this behavior.
           Since we do not have the power to recompile these, we
           emulate the SVr4 behavior. Sigh. */
        down_write(&current->mm->mmap_sem);
        error = do_mmap(NULL, 0, PAGE_SIZE, PROT_READ | PROT_EXEC,
                MAP_FIXED | MAP_PRIVATE, 0);
```

```
    up_write(&current->mm->mmap_sem);
}

#ifdef ELF_PLAT_INIT
/*
 * The ABI may specify that certain registers be set up in special
 * ways (on i386 %edx is the address of a DT_FINI function, for
 * example.  In addition, it may also specify (eg, PowerPC64 ELF)
 * that the e_entry field is the address of the function descriptor
 * for the startup routine, rather than the address of the startup
 * routine itself.  This macro performs whatever initialization to
 * the regs structure is required as well as any relocations to the
 * function descriptor entries when executing dynamically links apps.
 */
ELF_PLAT_INIT(regs, reloc_func_desc);
#endif

start_thread(regs, elf_entry, bprm->p);
if (unlikely(current->ptrace & PT_PTRACED)) {
    if (current->ptrace & PT_TRACE_EXEC)
        ptrace_notify ((PTRACE_EVENT_EXEC << 8) | SIGTRAP);
    else
        send_sig(SIGTRAP, current, 0);
}
retval = 0;
out:
kfree(loc);
out_ret:
return retval;

/* error cleanup */
out_free_dentry:
allow_write_access(interpreter);
if (interpreter)
    fput(interpreter);
out_free_interp:
kfree(elf_interpreter);
out_free_file:
sys_close(elf_exec_fileno);
out_free_fh:
if (files)
    reset_files_struct(current, files);
out_free_ph:
kfree(elf_phdata);
```

```
    goto out;
    }
```

load_elf_binary注释

```
static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs
*regs)
```

bprm 含有要载入的该可执行文件的必要信息，具体解释间 do_execve()的说明。
*regs 是用户态执行寄存器的表示。在内核态对该结构的任何修改，当该程序所代表的 process 切换到用户态执行时，会载入该结构中的内容。

```
    struct {
        struct elfhdr elf_ex;
        struct elfhdr interp_elf_ex;
            struct exec interp_ex;
    } *loc;

    loc = kmalloc(sizeof(*loc), GFP_KERNEL);
    if (!loc) {
        retval = -ENOMEM;
        goto out_ret;
    }
```

struct *loc 中的
struct elfhdr elf_ex 为 ELF 文件头的结构
struct elfhdr interp_elf_ex 为动态链接器的文件头的结构
struct exec interp_ex 是 aout 文件头的结构，由于这里分析的是 ELF 的载入过程，这里忽略。
分配该结构的内存。

```
    /* Get the exec-header */
    loc->elf_ex = *((struct elfhdr *)bprm->buf);
```

bprm->buf[]中包含有当前要载入的可执行文件的前 128 个 bytes。ELF header 即在里面。

```
    retval = -ENOEXEC;
    /* First of all, some simple consistency checks */
    if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
        goto out;
```

比较前 4 个 byte 是否是 ELF 格式文件的签名。

```
    if (loc->elf_ex.e_type != ET_EXEC && loc->elf_ex.e_type != ET_DYN)
        goto out;
```

比较要载入的可执行文件是什么类型的，只有 ET_EXEC(可执行文件)或 ET_SYN(共享库)才支

持。这里有个疑问，难道对载入动态链接库也要调用本函数吗？

```
    if (!elf_check_arch(&loc->elf_ex))
        goto out;
```

体系相关检查，在 x86 上检查是否是正确的 CPU。

```
    if (!bprm->file->f_op||!bprm->file->f_op->mmap)
        goto out;
```

可执行文件所在文件系统必须支持文件映射操作。

```
    /* Now read in all of the header information */
    if (loc->elf_ex.e_phentsize != sizeof(struct elf_phdr))
        goto out;
```

检查 ELF 文件中的 Program Header 的大小是否合法。

```
    if (loc->elf_ex.e_phnum < 1 ||
        loc->elf_ex.e_phnum > 65536U / sizeof(struct elf_phdr))
        goto out;
```

elf_ex.e_phnum 为 ELF 文件有几个 segment，这里对该可执行文件有多少个 segment 进行合法性检查。

```
    size = loc->elf_ex.e_phnum * sizeof(struct elf_phdr);
    retval = -ENOMEM;
    elf_phdata = kmalloc(size, GFP_KERNEL);
    if (!elf_phdata)
        goto out;
```

为了读入整个 Program Header Table，所以分配空间。

```
    retval = kernel_read(bprm->file, loc->elf_ex.e_phoff,
                (char *)elf_phdata, size);
    if (retval != size) {
        if (retval >= 0)
            retval = -EIO;
        goto out_free_ph;
    }
```

读入整个 Program Header Table。

```
    files = current->files;  /* Refcounted so ok */
    retval = unshare_files();
    if (retval < 0)
        goto out_free_ph;
    if (files == current->files) {
        put_files_struct(files);
        files = NULL;
```

```
    }

    /* exec will make our files private anyway, but for the a.out
       loader stuff we need to do it earlier */
    retval = get_unused_fd();
    if (retval < 0)
        goto out_free_fh;
```

取得一个未用的数字以用于作为 File Handle。

```
    get_file(bprm->file);


    fd_install(elf_exec_fileno = retval, bprm->file);
```

先把该数字赋值给 elf_exec_fileno，然后在 file table 中把该数字与 bprm->file 进行关联，这时候该数字才可称为 File Handle。而 bprm->file 为当前正在载入的可执行文件本身。

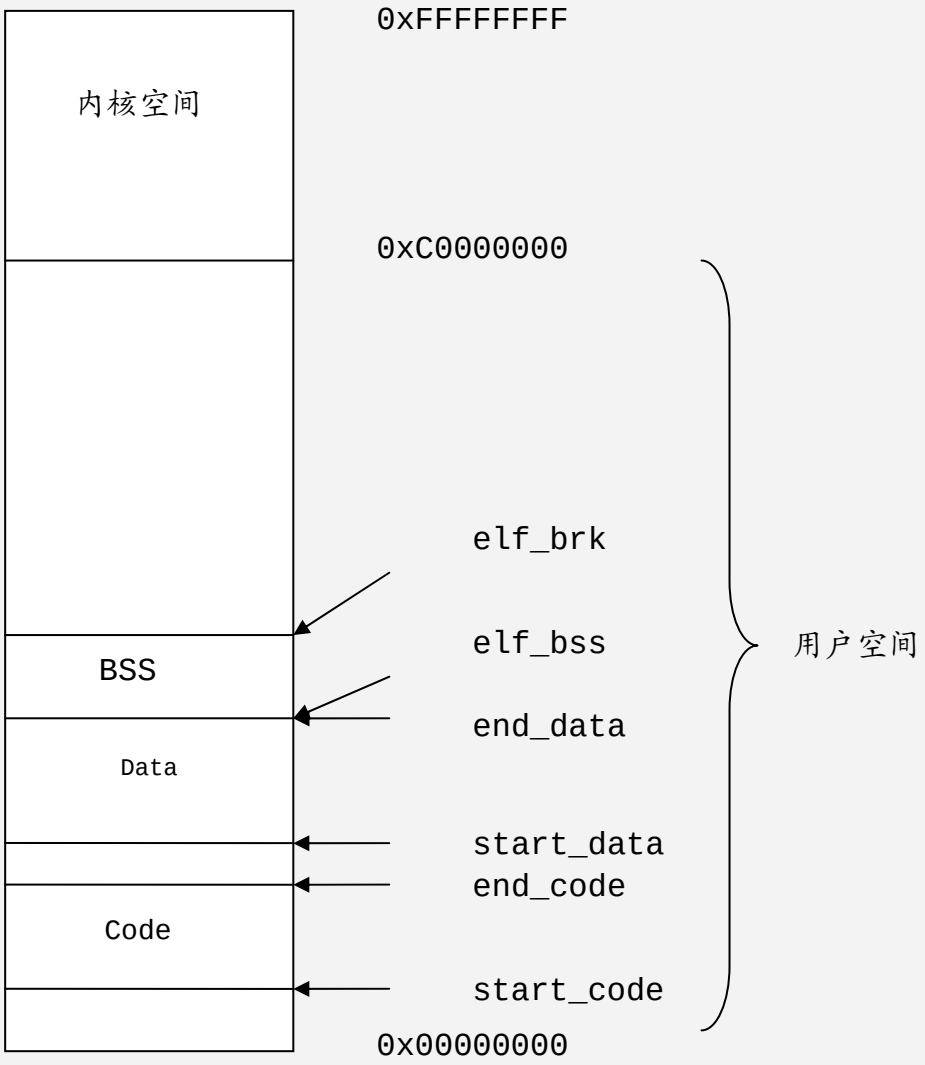
```
    elf_ppnt = elf_phdata;
    elf_bss = 0;
    elf_brk = 0;

    start_code = ~0UL;
    end_code = 0;
    start_data = 0;
    end_data = 0;
```

相关变量的初始化。

elf_bss 用于追踪 bss 段的首地址，elf_brk 用于追踪 bss 段的尾地址。elf_brk – elf_bss 就是整个 bss 段的大小，也是 ELF loader(也就是本函数)要初始化为零的空间大小。

start_code，end_code 用于追踪本可执行文件的代码段的首地址与尾地址。

start_data, end_data 用于追踪本可执行文件的数据段的首地址与尾地址。



解释一下上图，如果数据段在文件中的大小与其在内存中映射的大小不同（只可能前者小于后者），表示存在 bss 段。这里 end_data 与 elf_bss 是重合的，而从 elf_bss 到 elf_brk 的 BSS 段需要内核在这里初始化为零。在下面代码中有。并且 end_code 完全可能与 start_data 重合。

```
    for (i = 0; i < loc->elf_ex.e_phnum; i++) {
        if (elf_ppnt->p_type == PT_INTERP) {
            /* This is the program interpreter used for
             * shared libraries - for now assume that this
             * is an a.out format binary
             */
```

该 for 循环用来查找动态链接器的 path 所在的 Program Header。该 segment 非常简单，就是包含动态链接器的 path，比如/lib/ld-2.2.93.so。

```
            retval = -ENOEXEC;
            if (elf_ppnt->p_filesz > PATH_MAX ||
```

```
                elf_ppnt->p_filesz < 2)
                goto out_free_file;
```

elf_ppnt->p_filesz 即是该 segment 的大小，由于只包含的是动态链接器的 path，自然长度不应该超过最长的 path（4096），当然也不应该小与 2。

```
            retval = -ENOMEM;
            elf_interpreter = kmalloc(elf_ppnt->p_filesz,
                        GFP_KERNEL);
            if (!elf_interpreter)
                goto out_free_file;
```

为存放动态链接器的 path 分配空间。

```
            retval = kernel_read(bprm->file, elf_ppnt->p_offset,
                       elf_interpreter,
                       elf_ppnt->p_filesz);
            if (retval != elf_ppnt->p_filesz) {
                if (retval >= 0)
                    retval = -EIO;
                goto out_free_interp;
            }
```

elf_ppnt->p_offset 为该 path 在文件中的 offset，elf_ppnt->p_filesz 为 path 的长度。比如读取出“/lib/ld-2.2.93.so”到上面分配的 elf_interpreter 空间中。

```
            /* make sure path is NULL terminated */
            retval = -ENOEXEC;
            if (elf_interpreter[elf_ppnt->p_filesz - 1] != '\0')
                goto out_free_interp;
```

注释说明了意图。

```
            /* If the program interpreter is one of these two,
             * then assume an iBCS2 image. Otherwise assume
             * a native linux image.
             */
            if (strcmp(elf_interpreter,"/usr/lib/libc.so.1") == 0 ||
                strcmp(elf_interpreter,"/usr/lib/ld.so.1") == 0)
                ibcs2_interpreter = 1;
```

比较该动态链接器是否是特殊的。我也不知道所谓 iBCS2 image 是什麽。

```
            SET_PERSONALITY(loc->elf_ex, ibcs2_interpreter);
```

忽略，不知道干什麽。

```
            interpreter = open_exec(elf_interpreter);
            retval = PTR_ERR(interpreter);
            if (IS_ERR(interpreter))
```

```
                goto out_free_interp;
```
打开该动态链接器文件。

```
            /*
             * If the binary is not readable then enforce
             * mm->dumpable = 0 regardless of the interpreter's
             * permissions.
             */
            if (file_permission(interpreter, MAY_READ) < 0)
                bprm->interp_flags |= BINPRM_FLAGS_ENFORCE_NONDUMP;
```
不是很理解这里的代码。

```
            retval = kernel_read(interpreter, 0, bprm->buf,
                      BINPRM_BUF_SIZE);
            if (retval != BINPRM_BUF_SIZE) {
                if (retval >= 0)
                    retval = -EIO;
                goto out_free_dentry;
```
读出该动态链接器文件头上 128 个 byte。

```
            /* Get the exec headers */
            loc->interp_ex = *((struct exec *)bprm->buf);
            loc->interp_elf_ex = *((struct elfhdr *)bprm->buf);
            break;
```
设定指针，指向刚读出的那 128 个 byte。

```
    elf_ppnt = elf_phdata;
```
重新把 elf_ppnt 设到 Program Header Table 的头上。

```
    for (i = 0; i < loc->elf_ex.e_phnum; i++, elf_ppnt++)
        if (elf_ppnt->p_type == PT_GNU_STACK) {
            if (elf_ppnt->p_flags & PF_X)
                executable_stack = EXSTACK_ENABLE_X;
            else
                executable_stack = EXSTACK_DISABLE_X;
            break;
        }
```
枚举当前可执行文件的 Program Header，找到里面的 stack，根据该 segment 的属性来设置该 stack 是否可执行（execuatble）。可执行 stack 有一定的危险度，臭名昭著的 stack buffer overflow 攻击就是利用了可执行 stack 这个特性。这里 ELF 文件对 stack 提供了一定的控制度，可以屏蔽 stack 的可执行特性。

```
    /* Some simple consistency checks for the interpreter */
    if (elf_interpreter) {
```

```
        interpreter_type = INTERPRETER_ELF | INTERPRETER_AOUT;
```
如果存在动态链接器，则要做一些检查。
首先假设动态链接器类型未定，可以是 ELF 型或 aout 型。
这里我就有点不明白了，现在是在 ELF loader 中，而 aout loader 另由其他函数处理，为什么这里不能假设就是 ELF 型呢?

```
        /* Now figure out which format our binary is */
        if ((N_MAGIC(loc->interp_ex) != OMAGIC) &&
            (N_MAGIC(loc->interp_ex) != ZMAGIC) &&
            (N_MAGIC(loc->interp_ex) != QMAGIC))
            interpreter_type = INTERPRETER_ELF;
```
aout 型可执行文件有多种子类型，如 NMAGIC，OMAGIC，ZMAGIC，QMAGIC。如果不是这几种，就认为是 ELF 文件。

```
        if (memcmp(loc->interp_elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
            interpreter_type &= ~INTERPRETER_ELF;
```
在该文件头上是否有“ELF”的签名，如果没有，则不是 ELF 文件。

```
        retval = -ELIBBAD;
        if (!interpreter_type)
            goto out_free_dentry;
```
如果即不是 aout 型，又不是 ELF 型，则出错，退出。

```
        /* Make sure only one type was selected */
        if ((interpreter_type & INTERPRETER_ELF) &&
             interpreter_type != INTERPRETER_ELF) {
                // FIXME - ratelimit this before re-enabling
            // printk(KERN_WARNING "ELF: Ambiguous type, using ELF\n");
            interpreter_type = INTERPRETER_ELF;
        }
```
要么是 ELF，要么是 aout，不可能两者都是。

```
        /* Verify the interpreter has a valid arch */
        if ((interpreter_type == INTERPRETER_ELF) &&
            !elf_check_arch(&loc->interp_elf_ex))
            goto out_free_dentry;

#define elf_check_arch(x) \
    (((x)->e_machine == EM_386) || ((x)->e_machine == EM_486))

    /* OK, we are done with that, now set up the arg stuff,
       and then start this sucker up */
    if ((!bprm->sh_bang) && (interpreter_type == INTERPRETER_AOUT)) {
        char *passed_p = passed_fileno;
```

```
        sprintf(passed_fileno, "%d", elf_exec_fileno);

        if (elf_interpreter) {
            retval = copy_strings_kernel(1, &passed_p, bprm);
            if (retval)
                goto out_free_dentry;
            bprm->argc++;
        }
    }
```

bprm->sh_bang 表示该可执行文件是否是 script executable file，这里当然不是。并且该动态链接器也非 aout 型，所以不会执行这段代码。

```
    /* Flush all traces of the currently running executable */
    retval = flush_old_exec(bprm);
    if (retval)
        goto out_free_dentry;
```

flush_old_exec()会释放掉原来的地址空间。在调用该函数以前，本程序运行的地址空间是 fork 该进程的父进程的地址空间。从此开始，要开始崭新的生活了！

```
    /* Discard our unneeded old files struct */
    if (files) {
        put_files_struct(files);
        files = NULL;
    }
```

释放无用的 file 结构。

```
    /* OK, This is the point of no return */
    current->mm->start_data = 0;
    current->mm->end_data = 0;
    current->mm->end_code = 0;
    current->mm->mmap = NULL;
    current->flags &= ~PF_FORKNOEXEC;
    current->mm->def_flags = def_flags;
```

初始化相应成员。

mm->start_data 表示该程序的数据段在地址空间中的开始。

mm->end_data 表示该程序的数据段在地址空间中的结束。

mm->end_code 表示该程序的代码段在地址空间中的结束。

mm->mmap 是进程内存映射之根。

清除掉 PF_FORKNOEXEC 标志。该标志表示进程刚创建，但还没执行。当然现在不符合了。

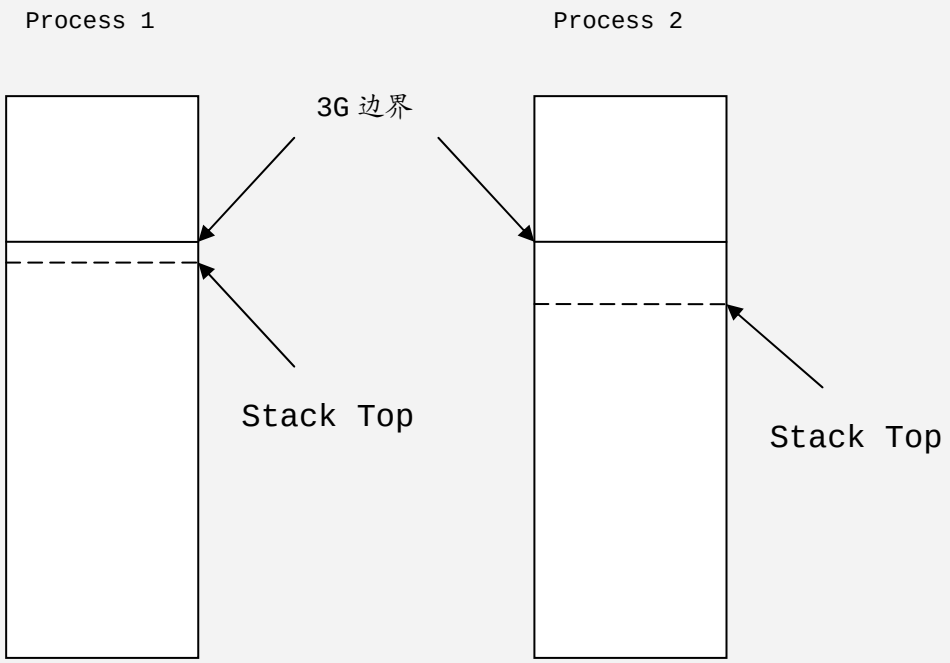
```
    /* Do this immediately, since STACK_TOP as used in setup_arg_pages
       may depend on the personality.  */
    SET_PERSONALITY(loc->elf_ex, ibcs2_interpreter);
```

在 x86 CPU 上该行为空，见下：

```
#define SET_PERSONALITY(ex, ibcs2) do { } while (0)

    if (elf_read_implies_exec(loc->elf_ex, executable_stack))
        current->personality |= READ_IMPLIES_EXEC;
#define elf_read_implies_exec
      (ex, executable_stack)(executable_stack != EXSTACK_DISABLE_X)
```
在 x86 CPU 上，可读即意味着可执行

```
    if     (!(current->personality     &     ADDR_NO_RANDOMIZE)     &&
randomize_va_space)
        current->flags |= PF_RANDOMIZE;
```
这里设置一些 personality，比较新引入，具体含义不是很清楚。

```
    arch_pick_mmap_layout(current->mm);


    /* Do this so that we can load the interpreter, if need be.  We will
       change some of these later */
    current->mm->free_area_cache = current->mm->mmap_base;
    current->mm->cached_hole_size = 0;

    retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                executable_stack);
    if (retval < 0) {
        send_sig(SIGKILL, current, 0);
```
这里 STACK_TOP 为 3G，即内核空间与用户空间的分界线。而 randomize_stack_top（STACK_TOP）的作用是使得栈顶的位置设置一定的随机化。如下:



在 3G 边界与 stack top 之间有一段随机大小的空隙

之所以要这样，可能是为了避免不同进程引用的栈变量的地址相同，因为这样 CPU cache 冲突率很高。具体 setup_arg_pages()函数的分析见该函数的单独分析。

```
        goto out_free_dentry;
    }
    current->mm->start_stack = bprm->p;
```

bprm->p 指向 argv[]，envp[]空间，具体解释请参阅对 copy_strings()函数的分析。

```
    /* Now we do a little grungy work by mmaping the ELF image into
       the correct location in memory.  At this point, we assume that
       the image should be loaded at fixed address, not at a variable
       address. */
    for(i = 0, elf_ppnt = elf_phdata;
        i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
        int elf_prot = 0, elf_flags;
        unsigned long k, vaddr;
```

对可执行文件中的 segment 进行枚举。比如

```
$ readelf -l /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x80498c0
```

```
There are 7 program headers, starting at offset 52


Program Headers:
  Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR          0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP        0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD          0x000000 0x08048000 0x08048000 0x10018 0x10018 R E 0x1000
  LOAD          0x010020 0x08059020 0x08059020 0x00454 0x007c8 RW  0x1000
  DYNAMIC       0x010220 0x08059220 0x08059220 0x000d8 0x000d8 RW  0x4
  NOTE          0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  GNU_EH_FRAME  0x00ffec 0x08057fec 0x08057fec 0x0002c 0x0002c R   0x4
```

例子中有 7 个 segment。

```
        if (elf_ppnt->p_type != PT_LOAD)
            continue;
```

看上面的例子，标有 LOAD 就是属于 PT_LOAD 型。只有这种类型的才需要被载入内存。上面第一个 LOAD 型 segment 是该程序的代码段，而第二个是数据段。

```
        if (unlikely (elf_brk > elf_bss)) {
```

这里 unlikely()是个宏，被定义为一个 gcc 的内迁扩展函数，主要用在条件分支判断的优化方面，你就把它看成空而已，对代码理解没有影响。

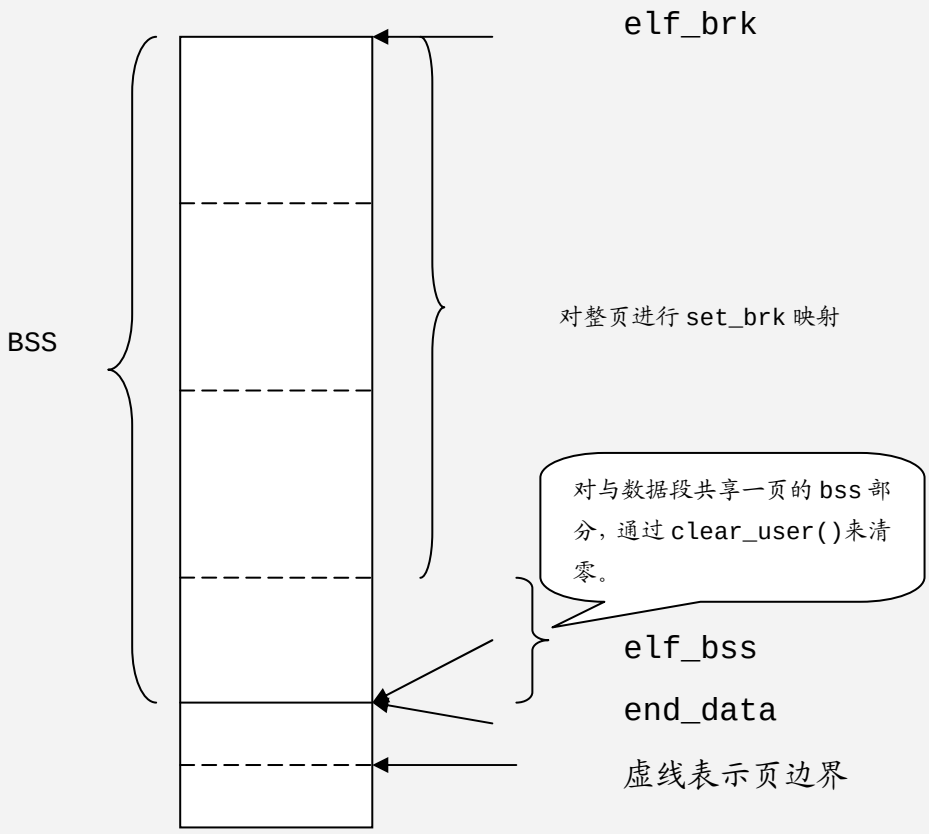
```
            unsigned long nbyte;

            /* There was a PT_LOAD segment with p_memsz > p_filesz
               before this one. Map anonymous pages, if needed,
               and clear the area.  */
            retval = set_brk (elf_bss + load_bias,
                    elf_brk + load_bias);
            if (retval) {
                send_sig(SIGKILL, current, 0);
                goto out_free_dentry;
            }
```

映射 BSS 段，并初始化零。这里由于是对页取整的，所以不包括 BSS 段与数据段在同一个 page 上的部分。

```
            nbyte = ELF_PAGEOFFSET(elf_bss);
            if (nbyte) {
                nbyte = ELF_MIN_ALIGN - nbyte;
                if (nbyte > elf_brk - elf_bss)
                    nbyte = elf_brk - elf_bss;
                if (clear_user((void __user *)elf_bss +
                           load_bias, nbyte)) {
```

```
                    /*
                     * This bss-zeroing can fail if the ELF
                     * file specifies odd protections. So
                     * we don't check the return value
                     */
                }
            }
        }
```

处理 BSS 段中与 data 段在同一 page 上的情况。见下图:




```
        if (elf_ppnt->p_flags & PF_R)
            elf_prot |= PROT_READ;
        if (elf_ppnt->p_flags & PF_W)
            elf_prot |= PROT_WRITE;
        if (elf_ppnt->p_flags & PF_X)
            elf_prot |= PROT_EXEC;
```

根据该 segment 在程序中的 Program Header 中的段属性来设置 File Mapping 属性。

```
        elf_flags = MAP_PRIVATE | MAP_DENYWRITE | MAP_EXECUTABLE;
```

如果察看 mmap() 系统调用的 manual，对 MAP_PRIVATE，MAP_DENYWRITE，

MAP_EXECUTABLE 的解释如下:

```
MAP_PRIVATE
     Create a private copy-on-write mapping.  Stores to the
     region do not affect the original file.  It is unspecified
     whether changes made to the file after the mmap call are
     visible in the mapped region.

MAP_DENYWRITE
     This flag is ignored.  (Long ago, it signalled that attempts to
     write to the underlying file should fail with ETXTBUSY.  But
     this was a source of denial-of-service attacks.)

MAP_EXECUTABLE
     This flag is ignored.

MAP_FIXED  Do not select a different address than the one specified.
     If the specified address cannot be used, mmap will fail.
     If MAP_FIXED is specified, start must be a multiple of the
     pagesize.  Use of this option is discouraged.
```

设置要映射的空间的属性。

```
       vaddr = elf_ppnt->p_vaddr;
```

elf_ppnt->p_vaddr 是指该段需要被映射到虚拟地址空间的那儿（具体地址）。比如

```
LOAD         0x000000 0x08048000 0x08048000 0x10018 0x10018 R E 0x1000
```

p_vaddr = 0x08048000，即该程序的代码段需要被映射到 0x08048000 的地址。对可执行文件而言，这个地址是必须满足的，它不象共享库可以被重定位。

```
       if (loc->elf_ex.e_type == ET_EXEC || load_addr_set) {
           elf_flags |= MAP_FIXED;
       } else if (loc->elf_ex.e_type == ET_DYN) {
           /* Try and get dynamic programs out of the way of the
            * default mmap base, as well as whatever program they
            * might try to exec.  This is because the brk will
            * follow the loader, and is not movable.  */
           load_bias = ELF_PAGESTART(ELF_ET_DYN_BASE - vaddr);
       }
```

ET_EXEC 表示该文件是可执行文件，ET_DYN 表示该文件是共享库。如果是可执行文件，则正象上面说的，映射时必须是固定地址的，即 mmap()程序没有自由来决定该映射到哪儿，因为设了 MAP_FIXED 标志；而如果是共享库，则 elf_ppnt->p_vaddr 只是相对于本模块基址的偏移(Offset)。比如:

```
$ readelf –l /lib/libc.so.6
Elf file type is DYN (Shared object file)
Entry point 0x159d0
There are 7 program headers, starting at offset 52
```

```
Program Headers:
  Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR          0x000034 0x00000034 0x00000034 0x000e0 0x000e0 R E 0x4
  INTERP        0x127d10 0x00127d10 0x00127d10 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD          0x000000 0x00000000 0x00000000 0x128068 0x128068 R E 0x1000
  LOAD          0x128080 0x00129080 0x00129080 0x0465c 0x08b84 RW  0x1000
  DYNAMIC       0x12c284 0x0012d284 0x0012d284 0x000d8 0x000d8 RW  0x4
  NOTE          0x000114 0x00000114 0x00000114 0x00020 0x00020 R   0x4
  GNU_EH_FRAME  0x127d24 0x00127d24 0x00127d24 0x00344 0x00344 R   0x4
```

上面共享库的代码段要求映射到 0x00000000 地址，而数据段要求映射到 0x00129080 地址，这当然是不可能的。这里的 VirtAddr 只是 Offset。比如如果/lib/libc.so.6 被载入到 0x40000000，则表示其代码段应该从 0x40000000 的 0x00000000 偏移开始，也就是从地址 0x40000000 开始；而其数据段应该从 0x40000000 的 0x00129080 偏移开始，也就是从地址 0x40129080 开始。

对共享库而言，#define ELF_ET_DYN_BASE (TASK_SIZE / 3 * 2)
即 3G / 3 * 2 = 2G，把共享库映射到地址空间 2G 以下。
上面对于

```
if (loc->elf_ex.e_type == ET_EXEC || load_addr_set) {
    elf_flags |= MAP_FIXED;
```

得判断可以轻易理解。直译就是如果是可执行文件(ET_EXEC)或者 load_addr_set 置位的情况下，文件映射必须满足固定地址映射，即 elf_map()函数的第二个参数不是推荐的映射地址，而是必须满足的映射地址，否则失败。这就是设置 MAP_FIXED 的效果。该 if 判断的第一个条件可以理解，因为可执行文件本来就不是可“浮动”的。而第二个条件 load_addr_set 就在紧接下面的代码出置位，它是针对共享库的非第一个 segment 映射的。在载入共享库时，第一个 segment(通常是该共享库的代码段)的映射地址就是该库被载入的基址，是可以“浮动”的。但一旦选定基址，则后面的 segment，比如该共享库的数据段，则没有自由了，它必须逻辑上挨着第一个 segment(之所以说是逻辑上，因为物理上当然可以中间不连续，有空洞等，但不能有其他外在模块的插入)，这样自然就必须是 fixed。所以第一个条件是针对可执行文件的，而第二个条件是针对共享库（ET_DYN）的。
else if 中的条件是针对共享库（ET_DYN）的第一个 segment 映射的，这里推荐的载入地址是 2G。
我这里有个疑问，在什么情况下，load_elf_binary()会被用来再加共享库呢？因为按常规，所有共享库都由 dynamic linker 在用户态载入。

```
error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt,
        elf_prot, elf_flags);
if (BAD_ADDR(error)) {
    send_sig(SIGKILL, current, 0);
    goto out_free_dentry;
}
```

进行 File Mapping，如果失败，则内核向当前进程发 SIGKILL 信号，也就是 kill 该进程。

我这里有点疑惑，这都到了这份上，如果文件映射失败，当该进程返回到用户态时，还能正确处理 SIGKULL 这个信号吗？因为这时候原来的继承自父进程的地址空间已经在上面运行 flush_old_exec()以后被彻底释放，而新的地址空间又正在建立时失败，那怎么运行 SIGKILL 信号的 handler 呢？当然该信号有点特殊，属于应用程序无法改写的信号。当然可以通过阅读内核关于 signal handling 的源代码来回答，但还有一个更简单，快捷的方法。用 KDB 在 elf_map()函数返回哪儿设断点，然后修改返回值，使得返回值指向非法空间，我这里设成 0xFFFFFFFF，代码将走出错分支，即设置 SIGKILL 信号，然后退出。在 console 界面上打印出 “Killed”。OK！内核能正确处理。

```
        if (!load_addr_set) {
            load_addr_set = 1;
            load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
            if (loc->elf_ex.e_type == ET_DYN) {
                load_bias += error -
                             ELF_PAGESTART(load_bias + vaddr);
                load_addr += load_bias;
                reloc_func_desc = load_bias;
            }
        }
```

只有第一次载入 segment 时才会进入该段代码。首先把 load_addr_set 置 1。elf_ppnt->p_offset 为该 segment 在文件中的偏移，而 elf_ppnt->p_vaddr 为该 segment 被载入后所在的虚拟地址。比如：

```
[wzhou@dcmp10 ~]$ readelf -l /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x8049d80
There are 8 program headers, starting at offset 52

Program Headers:
  Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR          0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP        0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD          0x000000 0x08048000 0x08048000 0x13040 0x13040 R E 0x1000
  LOAD          0x013040 0x0805c040 0x0805c040 0x01141 0x01141 RW  0x1000
  DYNAMIC       0x013480 0x0805c480 0x0805c480 0x000e0 0x000e0 RW  0x4
  NOTE          0x000148 0x08048148 0x08048148 0x00020 0x00020 R   0x4
  GNU_EH_FRAME  0x012f78 0x0805af78 0x0805af78 0x0002c 0x0002c R   0x4
  GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

上面标兰的是文件内偏移，而表红的则是该 segment 映射到的虚拟地址。

所以上面的 load_addr 对可执行文件而言是载入的基址。对共享库有点不一样。load_bias 是该 segment 世纪载入地址与希望载入地址（elf_map()的返回值 error 是实际载入地址，而 ELF_PAGESTART(load_bias + vaddr)是传递给 elf-map()的推荐地址。因为共享库的载入是“浮动”的，没有指定 MAP_FIXED 标志）。reloc_func_desc 变量纪录了该位移。

```
        k = elf_ppnt->p_vaddr;
        if (k < start_code)
            start_code = k;
        if (start_data < k)
            start_data = k;
```

在整个模块被载入后，start_code 总是追踪模块中地址映射的最低地址，而 start_data 总是指向位于最高地址的 segment 的开始。这短短几行代码，实际上隐含了对 ELF 格式的诸多假定，使得 ELF 格式理论上的“变化多端”荡然无存。

假设之一：代码段必须是可载入段的第一个段，否则 start_code 指向肯定错了。

假设之二：数据段必须是可载入段的最后一个段，否则 start_data 指向肯定错了。

其实我只要把常规可执行文件中的代码段与数据段换一下位置，该可执行文件虽然符合 ELF 格式，但 Linux 却拒绝载入。当然合格的 linker 也不会生成这样怪的可执行文件。但由此可以看出 linker 之类工具是如何的与 OS 紧密相连，谈不上什么 portable。

```
        if (BAD_ADDR(k) || elf_ppnt->p_filesz > elf_ppnt->p_memsz ||
            elf_ppnt->p_memsz > TASK_SIZE ||
            TASK_SIZE - elf_ppnt->p_memsz < k) {
            /* set_brk can never work. Avoid overflows. */
            send_sig(SIGKILL, current, 0);
            goto out_free_dentry;
        }
```

一些合法性检查。

```
#define BAD_ADDR(x) ((unsigned long)(x) >= TASK_SIZE)
```

k 是用户态地址，当然不能超过 3G(TASK_SIZE)。

```
elf_ppnt->p_filesz > elf_ppnt->p_memsz
```

该段在文件中的大小不能超过其在内存中的大小。当然不行，如果超过，不是表示有东西没映射进内存吗？

```
elf_ppnt->p_memsz > TASK_SIZE
```

一个段占的内存大小超过 3G，即整个用户空间。太大了吧，看来只有 64 位操作系统适合它了。

```
TASK_SIZE - elf_ppnt->p_memsz < k
```

即

```
k + elf_ppnt->p_memsz > TASK_SIZE
```

k 为该段被映射进内存后的首地址，该表达式表示该 segment 的尾巴超过了用户空间的边界，当然也不允许。

```
        k = elf_ppnt->p_vaddr + elf_ppnt->p_filesz;
```

k 为该段首址加上该段在文件中的大小。下面的代码

```
        k = elf_ppnt->p_vaddr + elf_ppnt->p_memsz;
```

k 为该段首址加上该段在内存中的大小。

对代码段而言，这两个 k 是完全是一样的。但对数据段而言，可能不一样。一般后者略大于前者，这多出来的空间就是未初始化的全局变量，也就是所谓的 bss 段。

```
        if (k > elf_bss)
```

```
            elf_bss = k;
```
记录 bss 段的起始。

```
        if ((elf_ppnt->p_flags & PF_X) && end_code < k)
            end_code = k;
```
变量 end_code 用于追踪可执行段的尾端

```
        if (end_data < k)
            end_data = k;
```
变量 end_data 用于追踪数据段的尾端。

```
        k = elf_ppnt->p_vaddr + elf_ppnt->p_memsz;
        if (k > elf_brk)
            elf_brk = k;
```
从 elf_bss 到 elf_brk 实际上就是 bss 段的大小。

```
    loc->elf_ex.e_entry += load_bias;
    elf_bss += load_bias;
    elf_brk += load_bias;
    start_code += load_bias;
    end_code += load_bias;
    start_data += load_bias;
    end_data += load_bias;
```
调整各个追踪变量。对可执行文件（ET_EXEC）而言，load_bias 为零，无所谓调整；而对共享库而言，则需要调整。

```
    /* Calling set_brk effectively mmaps the pages that we need
     * for the bss and break sections.  We must do this before
     * mapping in the interpreter, to make sure it doesn't wind
     * up getting placed where the bss needs to go.
     */
    retval = set_brk(elf_bss, elf_brk);
    if (retval) {
        send_sig(SIGKILL, current, 0);
        goto out_free_dentry;
    }

    if (likely(elf_bss != elf_brk) && unlikely(padzero(elf_bss))) {
        send_sig(SIGSEGV, current, 0);
        retval = -EFAULT; /* Nobody gets to see this, but.. */
        goto out_free_dentry;
    }
```
初始化 BSS 段。

```
    if (elf_interpreter) {
```
如果是动态链接的，则
```
        if (interpreter_type == INTERPRETER_AOUT)
            elf_entry = load_aout_interp(&loc->interp_ex,
                                interpreter);
        else
            elf_entry = load_elf_interp(&loc->interp_elf_ex,
                                interpreter,
                                &interp_load_addr);
        if (BAD_ADDR(elf_entry)) {
            force_sig(SIGSEGV, current);
            retval = IS_ERR((void *)elf_entry) ?
                    (int)elf_entry : -EINVAL;
            goto out_free_dentry;
        }
```
根据动态链接器的不同类型调用不同的函数来载入动态链接器。我不知道在什么情况下，ELF 格式的可执行文件链接的确是 aout 型的动态链接器。不解？ELF 型动态链接器的载入见 load_elf_interp()的分析。

返回值有两个，elf_entry 是动态链接器的入口，在本进程返回到用户态后要执行的第一条指令；interp_load_addr 是 ELF 格式的动态链接器被载入的基址。比如 0x40000000。
```
        reloc_func_desc = interp_load_addr;

        allow_write_access(interpreter);
        fput(interpreter);
        kfree(elf_interpreter);
    } else {
```
这里表示该可执行文件是静态链接的，没有什么动态链接的事，这就简单多了。
```
        elf_entry = loc->elf_ex.e_entry;
```
把该可执行文件的入口设为本进程返回到用户态后运行的第一条指令。
```
        if (BAD_ADDR(elf_entry)) {
            force_sig(SIGSEGV, current);
            retval = -EINVAL;
            goto out_free_dentry;
```
出错处理。这里比较有趣，如果可执行文件的入口地址非法，内核向本进程发 SIGSEGV 信号。
```
        }
    }

    kfree(elf_phdata);
```
释放可执行文件 ELF header 空间。

```
    if (interpreter_type != INTERPRETER_AOUT)
        sys_close(elf_exec_fileno);
```

如果动态链接器不是 aout 型，则关闭可执行文件本身所对应的 File Handle。这是一个很有趣的特性，即可执行文件正在运行时，并不保留有对自身的文件关联。这造成的后果是，可执行文件还在运行，但用户却可以成功的删除该可执行文件。比如:

我们可以写一个自删除的程序。

```
[wzhou@dcmp10 ~]$ cat remove-self.c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char** argv, char** env)
{
       if(unlink(argv[0]) == 0)
       {
              printf("delete self successfully!\n");
       }
       else
       {
              printf("delete self fail!\n");
       }

       getchar();
       return 0;
}
```

通过系统调用 unlink()来删除自身。返回值为零，表示删除成功。

加上 getchar()是为了让该程序在自删除以后不要马上退出，以便我们察看信息。

```
[wzhou@dcmp10 ~]$ ./remove-self &
[2] 2410
[wzhou@dcmp10 ~]$ delete self successfully!


[2]+  Stopped                 ./remove-self
[wzhou@dcmp10 ~]$
```

自删除成功，但该程序所代表的 process 应该还未退出。

```
[wzhou@dcmp10 ~]$ ps aux | grep remove-self
wzhou    2410  0.0  0.0  2052  284 pts/9    T    09:35   0:00 ./remove-self
wzhou    2414  0.0  0.0  4476  664 pts/9    S+   09:37   0:00 grep remove-self
```

标红的即是静态的程序已被删除，但动态的该程序所代表的 process 还在运行。

在 Windows 下你要写一个自删除的程序，可要许多奇技淫巧。因为本质上 Windows 是不允许自删除的，当它运行一个程序时，拥有该程序的 File Handle。

两种方式各有利弊。比如 Windows 下对普通用户的误删除文件可能好一点，但对反病毒软件杀毒却造成很大障碍。我在使用 Windows 中的一个大的烦恼就是打开反病毒软件实时监控后造成的系统低效，所以时常手工禁止该功能。但一旦可执行文件被感染，同时其又处于运行状态，反

病毒软件对它就束手无策。Microsoft 实在应该想一个办法，来解决这个问题。

```
    set_binfmt(&elf_format);
```

增加 ELF loader 所在模块的引用计数。

```
#ifdef ARCH_HAS_SETUP_ADDITIONAL_PAGES
    retval = arch_setup_additional_pages(bprm, executable_stack);
    if (retval < 0) {
        send_sig(SIGKILL, current, 0);
        goto out;
    }
#endif /* ARCH_HAS_SETUP_ADDITIONAL_PAGES */
```

这是一个与 CPU 架构相关的函数，对 i386 而言，是为 2.6 内核新引入的快速陷入内核做准备。主要是映射内核地址空间的最高一块作“sysenter return”之用。比如：

```
[wzhou@dcmp10 ~]$ cat /proc/17997/maps
00766000-0077b000 r-xp 00000000 fd:00 492674    /lib/ld-2.3.4.so
0077b000-0077c000 r--p 00014000 fd:00 492674    /lib/ld-2.3.4.so
0077c000-0077d000 rw-p 00015000 fd:00 492674    /lib/ld-2.3.4.so
0077f000-008a2000 r-xp 00000000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a2000-008a3000 r--p 00123000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a3000-008a6000 rw-p 00124000 fd:00 492914    /lib/tls/libc-2.3.4.so
008a6000-008a8000 rw-p 008a6000 00:00 0
08048000-08049000 r-xp 00000000 fd:01 378914    /home/wzhou/hello
08049000-0804a000 rw-p 00000000 fd:01 378914    /home/wzhou/hello
b7fec000-b7fed000 rw-p b7fec000 00:00 0
b7ffe000-b8000000 rw-p b7ffe000 00:00 0
bfe72000-c0000000 rw-p bfe72000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

也就是上面最后一行。

首先这个地址就很怪，0xffffe000 - 0xfffff000，这是整个 4G 地址空间的最后第二页。对 4G 地址空间的顶端页面，OS 有些特殊的指定，见 src/include/asm-i386/fixmap.h

```
#ifndef CONFIG_COMPAT_VDSO
extern unsigned long __FIXADDR_TOP;
#else
#define __FIXADDR_TOP  0xfffff000
#endif

。。。
enum fixed_addresses {
    FIX_HOLE,       最后第一页
    FIX_VDSO,       最后第二页
#ifdef CONFIG_X86_LOCAL_APIC
    FIX_APIC_BASE,  /* local (CPU) APIC) -- required for SMP or not */
#endif
#ifdef CONFIG_X86_IO_APIC
```

```
    FIX_IO_APIC_BASE_0,
    FIX_IO_APIC_BASE_END = FIX_IO_APIC_BASE_0 + MAX_IO_APICS-1,
#endif
#ifdef CONFIG_X86_VISWS_APIC
    FIX_CO_CPU, /* Cobalt timer */
    FIX_CO_APIC,    /* Cobalt APIC Redirection Table */
    FIX_LI_PCIA,    /* Lithium PCI Bridge A */
    FIX_LI_PCIB,    /* Lithium PCI Bridge B */
#endif
#ifdef CONFIG_X86_F00F_BUG
    这个页面在我的文章《CPU Bug与Linux Kernel》一文中介绍过
    FIX_F00F_IDT,   /* Virtual mapping for IDT */
。。。
#endif
```

在src/arch/i386/kernel/sysenter.c中有如下代码

```
int __init sysenter_setup(void)
{
    向物理页管理器申请一页已初始化为零的物理页
    syscall_page = (void *)get_zeroed_page(GFP_ATOMIC);

#ifdef CONFIG_COMPAT_VDSO
    __set_fixmap(FIX_VDSO, __pa(syscall_page), PAGE_READONLY);
    printk("Compat vDSO mapped to %08lx.\n", __fix_to_virt(FIX_VDSO));
#else
    /*
     * In the non-compat case the ELF coredumping code needs the fixmap:
     */
    给刚才分配的物理页建立地址映射关系，即为其建立page entry，虚拟地址为0xffffe000
    在所有进程的地址空间都能看到这一页。
    __set_fixmap(FIX_VDSO, __pa(syscall_page), PAGE_KERNEL_RO);
#endif

    if (!boot_cpu_has(X86_FEATURE_SEP)) {
    如果不利用快速系统调用，则拷贝int 0x80方式的老的系统调用方法的代码到0xffffe000
        memcpy(syscall_page,
               &vsyscall_int80_start,
               &vsyscall_int80_end - &vsyscall_int80_start);
        return 0;
    }
    利用快速系统调用，则拷贝代码到0xffffe000。代码见下面。
    memcpy(syscall_page,
          &vsyscall_sysenter_start,
          &vsyscall_sysenter_end - &vsyscall_sysenter_start);
```

```
    return 0;
}
```

所谓快速陷入内核是相对于以前通过发 int 0x80 软件中断来陷入内核的普通方式而言。具体内容请参考 intel manual。（intel 官方网站能免费下载到）。优点自然是象其名字一样，“快速”。该 page 的内容在源代码里面的 src/arch/i386/kernel/vsyscall-sysenter.S 中，也就是上面拷贝到 0xffffe000 页面中的代码。

```
    .text
    .globl __kernel_vsyscall
    .type __kernel_vsyscall,@function
__kernel_vsyscall:
.LSTART_vsyscall:
    push %ecx
.Lpush_ecx:
    push %edx
.Lpush_edx:
    push %ebp
.Lenter_kernel:
    movl %esp,%ebp
    sysenter                    这就是快速陷入指令

    /* 7: align return point with nop's to make disassembly easier */
    .space 7,0x90               nop 指令

    /* 14: System call restart point is here! (SYSENTER_RETURN-2) */
    jmp .Lenter_kernel
    /* 16: System call normal return point is here! */
    .globl SYSENTER_RETURN  /* Symbol used by sysenter.c  */
SYSENTER_RETURN:
    pop %ebp
.Lpop_ebp:
    pop %edx
.Lpop_edx:
    pop %ecx
.Lpop_ecx:
    ret
.LEND_vsyscall:
    .size __kernel_vsyscall,.-.LSTART_vsyscall
    .previous
```

具体 int 0x80 与 sysenter 之间差别及在 Linux 内核中的代码实现不再这里分析了。

```
    compute_creds(bprm);
```

```
    current->flags &= ~PF_FORKNOEXEC;
```
这个 flag 好象在前面的代码中已经 clear 过了。这里怎么又来一次。
PF_FORKNOEXEC 表示 fork 但还没有 execute，当然现在不符现状了。

```
    create_elf_tables(bprm, &loc->elf_ex,
             (interpreter_type == INTERPRETER_AOUT),
             load_addr, interp_load_addr);
```
见对 create_elf_tables（）的分析。

```
    /* N.B. passed_fileno might not be initialized? */
    if (interpreter_type == INTERPRETER_AOUT)
        current->mm->arg_start += strlen(passed_fileno) + 1;
    current->mm->end_code = end_code;
    current->mm->start_code = start_code;
    current->mm->start_data = start_data;
    current->mm->end_data = end_data;
    current->mm->start_stack = bprm->p;
```
设置 task_struct 结构(管理 process 的最核心结构)中用户态虚拟内存管理的相应变量。

```
    if (current->personality & MMAP_PAGE_ZERO) {
        /* Why this, you ask???  Well SVr4 maps page 0 as read-only,
           and some applications "depend" upon this behavior.
           Since we do not have the power to recompile these, we
           emulate the SVr4 behavior. Sigh. */
        down_write(&current->mm->mmap_sem);
        error = do_mmap(NULL, 0, PAGE_SIZE, PROT_READ | PROT_EXEC,
                MAP_FIXED | MAP_PRIVATE, 0);
        up_write(&current->mm->mmap_sem);
    }
```
从上面代码中的注释看，好像 SVr4 有要求，把地址空间的首页变成 NULL page。显然是为了捕捉 NULL pointer。不过我也没用过 SVr4 型的 Unix OS。

```
#ifdef ELF_PLAT_INIT
    /*
     * The ABI may specify that certain registers be set up in special
     * ways (on i386 %edx is the address of a DT_FINI function, for
     * example.  In addition, it may also specify (eg, PowerPC64 ELF)
     * that the e_entry field is the address of the function descriptor
     * for the startup routine, rather than the address of the startup
     * routine itself.  This macro performs whatever initialization to
     * the regs structure is required as well as any relocations to the
     * function descriptor entries when executing dynamically links apps.
     */
    ELF_PLAT_INIT(regs, reloc_func_desc);
```

```
#endif
    start_thread(regs, elf_entry, bprm->p);
    ELF_PLAT_INIT 是 macro，定义如下:
    #define ELF_PLAT_INIT(_r, load_addr)   do { \
    _r->ebx = 0; _r->ecx = 0; _r->edx = 0; \
    _r->esi = 0; _r->edi = 0; _r->ebp = 0; \
    _r->eax = 0; \
    } while (0)
    start_thread 是个 macro，定义如下:
    #define start_thread(regs, new_eip, new_esp) do {    \
    __asm__("movl %0,%%fs": :"r" (0));        \
    regs->xgs = 0;                    \
    set_fs(USER_DS);                \
    regs->xds = __USER_DS;                \
    regs->xes = __USER_DS;                \
    regs->xss = __USER_DS;                \
    regs->xcs = __USER_CS;                \
    regs->eip = new_eip;                \
    regs->esp = new_esp;                \
    } while (0)

    这里实际上是设置该可执行文件最初始执行时的寄存器的状态。可见在 Linux 下可执行文
件执行第一条指令时的 CPU 状态是这样的:

    eax = 0
    ebx = 0
    ecx = 0
    edx = 0
    esi = 0
    edi = 0
    ebp = 0
    gs = 0
    ds = 15 * 8 + 3 = 123
    es = 123
    ss = 123
    cs = 14 * 8 + 3 = 115
    如果是静态链接的可执行文件
    eip = 可执行文件本身的入口
    如果是动态链接的可执行文件
    eip = 动态链接器的入口
    指向用户空间顶端分配给 argv 与环境变量的空间，前面有介绍。

    if (unlikely(current->ptrace & PT_PTRACED)) {
        if (current->ptrace & PT_TRACE_EXEC)
```

```
            ptrace_notify ((PTRACE_EVENT_EXEC << 8) | SIGTRAP);
        else
            send_sig(SIGTRAP, current, 0);
    }
    retval = 0;
```

current->ptrace & PT_PTRACED 表示当前进程正在被调试，也即 debugger “exec”了该执行文件。如果 PT_TRACE_EXEC 同样置位的话，要特殊处理（？）PT_TRACE_EXEC 好像是一个比较新的标志，没研究过在这里是干什么的。否则就发送 SIGTRAP signal，而 debugger 会捕捉该信号，来实现调试。

load_elf_interp源码

```
/* This is much more generalized than the library routine read function,
   so we keep this separate.  Technically the library read function
   is only provided so that we can read a.out libraries that have
   an ELF header */

static unsigned long load_elf_interp(struct elfhdr *interp_elf_ex,
        struct file *interpreter, unsigned long *interp_load_addr)
{
    struct elf_phdr *elf_phdata;
    struct elf_phdr *eppnt;
    unsigned long load_addr = 0;
    int load_addr_set = 0;
    unsigned long last_bss = 0, elf_bss = 0;
    unsigned long error = ~0UL;
    int retval, i, size;

    /* First of all, some simple consistency checks */
    if (interp_elf_ex->e_type != ET_EXEC &&
        interp_elf_ex->e_type != ET_DYN)
        goto out;
    if (!elf_check_arch(interp_elf_ex))
        goto out;
    if (!interpreter->f_op || !interpreter->f_op->mmap)
        goto out;

    /*
     * If the size of this structure has changed, then punt, since
     * we will be doing the wrong thing.
     */
    if (interp_elf_ex->e_phentsize != sizeof(struct elf_phdr))
        goto out;
    if (interp_elf_ex->e_phnum < 1 ||
        interp_elf_ex->e_phnum > 65536U / sizeof(struct elf_phdr))
        goto out;

    /* Now read in all of the header information */
    size = sizeof(struct elf_phdr) * interp_elf_ex->e_phnum;
    if (size > ELF_MIN_ALIGN)
        goto out;
    elf_phdata = kmalloc(size, GFP_KERNEL);
    if (!elf_phdata)
```

```
        goto out;

    retval = kernel_read(interpreter, interp_elf_ex->e_phoff,
                (char *)elf_phdata,size);
    error = -EIO;
    if (retval != size) {
        if (retval < 0)
            error = retval;
        goto out_close;
    }

    eppnt = elf_phdata;
    for (i = 0; i < interp_elf_ex->e_phnum; i++, eppnt++) {
        if (eppnt->p_type == PT_LOAD) {
            int elf_type = MAP_PRIVATE | MAP_DENYWRITE;
            int elf_prot = 0;
            unsigned long vaddr = 0;
            unsigned long k, map_addr;

            if (eppnt->p_flags & PF_R)
                elf_prot = PROT_READ;
            if (eppnt->p_flags & PF_W)
                elf_prot |= PROT_WRITE;
            if (eppnt->p_flags & PF_X)
                elf_prot |= PROT_EXEC;
            vaddr = eppnt->p_vaddr;
            if (interp_elf_ex->e_type == ET_EXEC || load_addr_set)
                elf_type |= MAP_FIXED;

            map_addr = elf_map(interpreter, load_addr + vaddr,
                        eppnt, elf_prot, elf_type);
            error = map_addr;
            if (BAD_ADDR(map_addr))
                goto out_close;

            if (!load_addr_set &&
                interp_elf_ex->e_type == ET_DYN) {
                load_addr = map_addr - ELF_PAGESTART(vaddr);
                load_addr_set = 1;
            }

            /*
             * Check to see if the section's size will overflow the
             * allowed task size. Note that p_filesz must always be
```

```
             * <= p_memsize so it's only necessary to check p_memsz.
             */
            k = load_addr + eppnt->p_vaddr;
            if (BAD_ADDR(k) ||
                eppnt->p_filesz > eppnt->p_memsz ||
                eppnt->p_memsz > TASK_SIZE ||
                TASK_SIZE - eppnt->p_memsz < k) {
                error = -ENOMEM;
                goto out_close;
            }

            /*
             * Find the end of the file mapping for this phdr, and
             * keep track of the largest address we see for this.
             */
            k = load_addr + eppnt->p_vaddr + eppnt->p_filesz;
            if (k > elf_bss)
                elf_bss = k;

            /*
             * Do the same thing for the memory mapping - between
             * elf_bss and last_bss is the bss section.
             */
            k = load_addr + eppnt->p_memsz + eppnt->p_vaddr;
            if (k > last_bss)
                last_bss = k;
        }
    }

    /*
     * Now fill out the bss section.  First pad the last page up
     * to the page boundary, and then perform a mmap to make sure
     * that there are zero-mapped pages up to and including the
     * last bss page.
     */
    if (padzero(elf_bss)) {
        error = -EFAULT;
        goto out_close;
    }

    /* What we have mapped so far */
    elf_bss = ELF_PAGESTART(elf_bss + ELF_MIN_ALIGN - 1);

    /* Map the last of the bss segment */
```

```
    if (last_bss > elf_bss) {
        down_write(&current->mm->mmap_sem);
        error = do_brk(elf_bss, last_bss - elf_bss);
        up_write(&current->mm->mmap_sem);
        if (BAD_ADDR(error))
            goto out_close;
    }

    *interp_load_addr = load_addr;
    error = ((unsigned long)interp_elf_ex->e_entry) + load_addr;

out_close:
    kfree(elf_phdata);
out:
    return error;
}
```

load_elf_interp注释

```
    static unsigned long load_elf_interp(struct elfhdr *interp_elf_ex,
        struct file *interpreter, unsigned long *interp_load_addr)
```

interp_elf_ex 指向动态链接器（/lib/ld-2.X.so）的 ELF header
interpreter 动态链接器文件指针
*interp_load_addr 返回动态链接器载入内存后的基地址

```
    if (interp_elf_ex->e_type != ET_EXEC &&
        interp_elf_ex->e_type != ET_DYN)
        goto out;
```

ET_EXEC 表示该动态链接器是可执行文件，类似 Windows 中的 exe 文件。
ET_DYN 表示该动态链接器是共享库，.so 文件。
如果不是上面两种，则出错退出。
这里对 ET_EXEC 的判断有点疑问，难道动态链接器还可以是单独以可执行文件形式出现吗？

```
    if (!elf_check_arch(interp_elf_ex))
        goto out;
```

与体系相关的检查，在 x86 CPU 上为

```
    #define elf_check_arch(x) \
    (((x)->e_machine == EM_386) || ((x)->e_machine == EM_486))
```

很显然是避免在错误的体系上去执行

```
    if (!interpreter->f_op || !interpreter->f_op->mmap)
        goto out;
```

检查动态链接器所在的文件系统是否支持“memory map”，文件映射操作。如果不支持，则出错退出。因为把动态链接器载入内存执行，完全依赖于相应文件系统的文件映射操作。几乎现代文件系统都支持该操作。

```
    if (interp_elf_ex->e_phentsize != sizeof(struct elf_phdr))
        goto out;
```

该动态链接器文件的 ELF header 的大小是否合法。

```
    if (interp_elf_ex->e_phnum < 1 ||
        interp_elf_ex->e_phnum > 65536U / sizeof(struct elf_phdr))
        goto out;
```

e_phnum 纪录了该 ELF 文件（动态链接器）有多少个 segment。这里检查该值是否在合理值之内。

```
    /* Now read in all of the header information */
    size = sizeof(struct elf_phdr) * interp_elf_ex->e_phnum;
    if (size > ELF_MIN_ALIGN)
```

```
        goto out;
```

struct elf_phdr 是 segment entry，给个 segment 都有一个，这里 size 就是所有 segment 的 entry size 总和。

```
#define ELF_MIN_ALIGN    PAGE_SIZE
```

size 为什么不能超过 4K，也就是 4096，在 ELF Format 上有记载吗？好像没看到。
size > ELF_MIN_ALIGN 只能说明这个动态链接器有超乎寻常多的 segment，只是怪而已，并不一定是错啊。
下面是某个版本的动态链接器的表示:

```
$readelf –l /lib/ld-2.2.93.so

Elf file type is DYN (Shared object file)
Entry point 0xb30
There are 3 program headers, starting at offset 52

Program Headers:
  Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD          0x000000 0x00000000 0x00000000 0x11da9 0x11da9 R E 0x1000
  LOAD          0x012000 0x00012000 0x00012000 0x00504 0x006e0 RW  0x1000
  DYNAMIC       0x012408 0x00012408 0x00012408 0x000b0 0x000b0 RW  0x4

 Section to Segment mapping:
  Segment Sections...

00     .hash .dynsym .dynstr .gnu.version .gnu.version_d .rel.dyn .rel.plt .plt .text .
rodata
   01     .data .dynamic .got .bss
   02     .dynamic
```

上面 DYN 是 ET_DYN 的表示。LOAD 是 PT_LOAD 的表示。第一个 segment 是动态链接器的盗码段，而第二个是数据段。上面的代码中只关心该两个 segment(有 LOAD 类型)。下面的说明我将用该动态链接器来举例说明。

```
    elf_phdata = kmalloc(size, GFP_KERNEL);
    if (!elf_phdata)
        goto out;
```

要读取所有 segment header，所以这里分配内存。

```
    retval = kernel_read(interpreter, interp_elf_ex->e_phoff,
                (char *)elf_phdata,size);
    error = -EIO;
    if (retval != size) {
        if (retval < 0)
```

```
            error = retval;
        goto out_close;
    }
```

从该动态链接器文件中读出所有 segment header。

```
    eppnt = elf_phdata;
    for (i = 0; i < interp_elf_ex->e_phnum; i++, eppnt++) {
        if (eppnt->p_type == PT_LOAD) {
```

下面是对动态链接器文件中所有 segment 的枚举，但这里只关心具有 PT_LOAD 类性的 segment。PT_LOAD 标志着该 segment 可以被载入内存。

```
            int elf_type = MAP_PRIVATE | MAP_DENYWRITE;
```

MAP_PRIVATE 表示该映射不能被共享。

```
            int elf_prot = 0;
            unsigned long vaddr = 0;
            unsigned long k, map_addr;

            if (eppnt->p_flags & PF_R)
                elf_prot = PROT_READ;
            if (eppnt->p_flags & PF_W)
                elf_prot |= PROT_WRITE;
            if (eppnt->p_flags & PF_X)
                elf_prot |= PROT_EXEC;
```

根据每个 segment 的属性来设置在文件映射(File Mapping)时内存的属性。

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x11da9 0x11da9 R E 0x1000
  LOAD           0x012000 0x00012000 0x00012000 0x00504 0x006e0 RW  0x1000
  DYNAMIC        0x012408 0x00012408 0x00012408 0x000b0 0x000b0 RW  0x4
```

上面标红的即是该 segment 的属性。对应到代码就是

```
R   ---     Readable    ---     PF_R
W   ---     Writable    ---     PF_W
E   ---     Executable  ---     PF_X

            if (interp_elf_ex->e_type == ET_EXEC || load_addr_set)
                elf_type |= MAP_FIXED;
```

代码只有两行，可解释起来有点麻烦。先分开来说。

```
interp_elf_ex->e_type == ET_EXEC
```

如果该动态链接器是可执行文件，而非共享库(ET_DYN)。这个条件实在有点莫名其妙。到目前的 Linux 系统为止，当然不满足该条件。所以上面的代码完全可以改成如下:

```
            if (load_addr_set)
                elf_type |= MAP_FIXED;
```

load_addr_set 在本函数的前面被初始化为零。

```
int load_addr_set = 0;
```

该变量是与 unsigned long load_addr 变量是联系在一起的。下面解释。

```
elf_type |= MAP_FIXED
```

MAP_FIXED 表示固定地址映射。函数

```
static unsigned long elf_map (
struct file *,
unsigned long,
struct elf_phdr *,
int,
int
);
```

的第二个参数是指定映射地址的。一般而言，这是个非强制性映射地址，如果已被占用，则会选择一个其他的地址，在 elf_map（）的返回值中返回。但如果指定了 MAP_FIXED，则必须映射到第二个参数指定的地址，否则就失败。

有上面的例子可看到，/lib/ld-2.2.93.so 有两个 LOAD 类型的 segment。

```
  LOAD           0x000000 0x00000000 0x00000000 0x11da9 0x11da9 R E 0x1000
  LOAD           0x012000 0x00012000 0x00012000 0x00504 0x006e0 RW  0x1000
```

当映射第一个 segment 时，load_addr_set 还是 0，所以在调用 elf_map（）时不会带 MAP_FIXED 属性，这样容许系统在选择该 segment 的映射地址上有一定的自由度。但一旦第一个 segment 的映射地址决定后，load_addr_set 被赋值成 1，这样第二个 segment 的映射地址就没有自由了。比如说第一个 segment 被映射到 0x40000000，系统也可以映射到 0x50000000，有一定自由度；但第二个 segment 则没有选择了，它必须跟着第一个 segment 的映射地址。因为第一个 segment 是代码段，第二个 segment 是数据段，它们两者之间是有关系的，不是各归各，不搭界的。可重定位的共享库不关心它被载入的绝对地址，但它关心内部 segment 间的相对位置。自由只是对映射时的第一个 segment 而言的。没有绝对的自由。

```
            map_addr = elf_map(interpreter, load_addr + vaddr,
                    eppnt, elf_prot, elf_type);
            error = map_addr;
            if (BAD_ADDR(map_addr))
                goto out_close;
```

这就是完成把动态链接器文件中的某个 segment 映射到某段地址空间了。

```
#cat /proc/356/maps
08048000-0804a000 r-xp 00000000 03:03 549481    /sbin/mingetty
0804a000-0804b000 rw-p 00002000 03:03 549481    /sbin/mingetty
0804b000-0804c000 rwxp 00000000 00:00 0
40000000-40012000 r-xp 00000000 03:03 808025    /lib/ld-2.2.93.so
```

```
40012000-40013000 rw-p 00012000 03:03 808025    /lib/ld-2.2.93.so
40014000-40015000 rw-p 00001000 00:00 0
4001b000-4001c000 rw-p 00000000 00:00 0
42000000-42126000 r-xp 00000000 03:03 581786    /lib/i686/libc-2.2.93.so
42126000-4212b000 rw-p 00126000 03:03 581786    /lib/i686/libc-2.2.93.so
4212b000-4212f000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
```

上面的例子就是文件映射的结果。可以看到/lib/ld-2.2.93.so 的代码段被映射到40000000-40012000，而数据段则被映射到 40012000-40013000。

```
            if (!load_addr_set &&
                interp_elf_ex->e_type == ET_DYN) {
                load_addr = map_addr - ELF_PAGESTART(vaddr);
                load_addr_set = 1;
            }
```

由于 interp_elf_ex->e_type == ET_DYN 肯定成立，可以简化为

```
            if (!load_addr_set) {
                load_addr = map_addr - ELF_PAGESTART(vaddr);
                load_addr_set = 1;
            }
```

在映射第一个 segment 时，load_addr_set 为 0，所以会执行

```
    load_addr = map_addr - ELF_PAGESTART(vaddr);
```

vaddr = eppnt->p_vaddr，即当前 segment 在文件中记录的映射地址。比如/lib/ld-2.2.93.so:

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x11da9 0x11da9 R E 0x1000
  LOAD           0x012000 0x00012000 0x00012000 0x00504 0x006e0 RW  0x1000
```

(当然实际上不可能真的映射到该地址)

map_addr 是 elf_map()函数的返回值，也是该 segment 真正的映射地址。很显然 load_addr = map_addr - ELF_PAGESTART(vaddr)后的 load_addr 是该模块的实际映射地址与理论映射地址间的 offset。又由于一般共享库的理论映射基址为零，所以 load_addr 也可以说是共享库的载入基址。

```
            k = load_addr + eppnt->p_vaddr;
            if (BAD_ADDR(k) ①||
                eppnt->p_filesz > eppnt->p_memsz ②||
                eppnt->p_memsz > TASK_SIZE ③||
                TASK_SIZE - eppnt->p_memsz < k)④ {
                error = -ENOMEM;
                goto out_close;
            }
```

K 为当前 segment 的映射首地址。if()中的任一条件满足都表示出错了。

①BAD_ADDR(k)
#define BAD_ADDR(x) ((unsigned long)(x) >= TASK_SIZE)
TASK_SIZE = 3G，k 超过 3G，即进入 kernel 空间了，当然出错。

②eppnt->p_filesz > eppnt->p_memsz
该 segment 在文件里的大小竟然比在内存里映射的空间大。当然错。等于或反之则对。

③eppnt->p_memsz > TASK_SIZE
该 segment 在内存里映射的空间竟然超过 3G。不可思议。

④TASK_SIZE - eppnt->p_memsz < k
即 k + eppnt->p_memsz > TASK_SIZE
如果这样，该 segment 溢出用户空间(user space)。

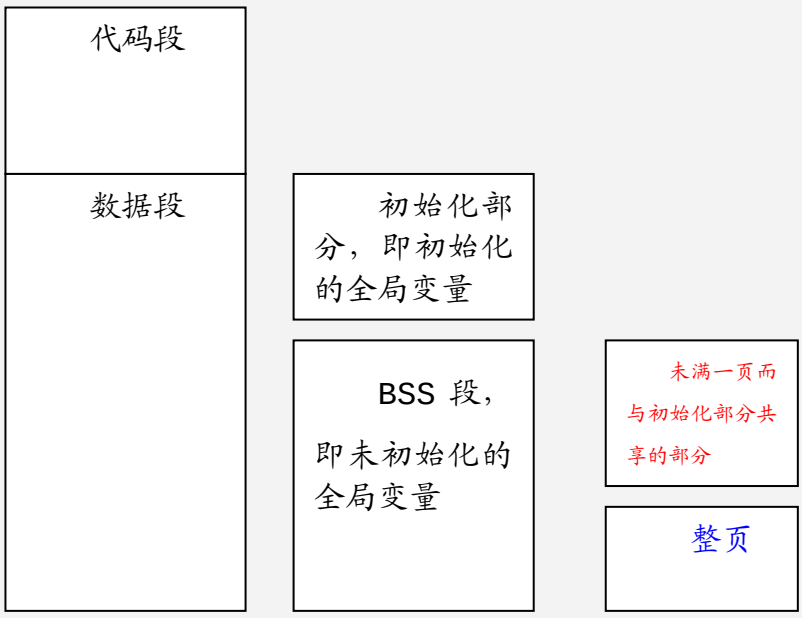
```
            /*
             * Find the end of the file mapping for this phdr, and
             * keep track of the largest address we see for this.
             */
            k = load_addr + eppnt->p_vaddr + eppnt->p_filesz;
            if (k > elf_bss)
                elf_bss = k;

            /*
             * Do the same thing for the memory mapping - between
             * elf_bss and last_bss is the bss section.
             */
            k = load_addr + eppnt->p_memsz + eppnt->p_vaddr;
            if (k > last_bss)
                last_bss = k;
        }
    }
```

eppnt->p_filesz 是该 segment 在文件内的大小，而 eppnt->p_memsz 是该 segment 在内存内的大小。eppnt->p_memsz >= eppnt->p_filesz。比如数据段，由于 bss 段是不占用文件空间的，但载入内存后，自然要为 bss 段分配空间，所以会造成不等。这里 elf_bss 用来跟踪该 segment 在内存中相对文件中大小的边界，last_bss 则是该 segment 实际在内存中的边界。把这段代码写在循环中，好像没必要。因为这里实际上已经隐含着数据段必须是 Program Header 的可载入段的最后一个 segment。如果在原始文件中 Program Header 表中先是数据段，然后是代码段，这段逻辑马上不对。
last_bss 与 elf_bss 之间的空间即是 bss 段大小，也就是程序中未初始化全局变量的大小。

```
    /*
     * Now fill out the bss section.  First pad the last page up
```

```
     * to the page boundary, and then perform a mmap to make sure
     * that there are zero-mapped pages up to and including the
     * last bss page.
     */
    if (padzero(elf_bss)) {
        error = -EFAULT;
        goto out_close;
    }
    /* What we have mapped so far */
    elf_bss = ELF_PAGESTART(elf_bss + ELF_MIN_ALIGN - 1);

    /* Map the last of the bss segment */
    if (last_bss > elf_bss) {
        down_write(&current->mm->mmap_sem);
        error = do_brk(elf_bss, last_bss - elf_bss);
        up_write(&current->mm->mmap_sem);
        if (BAD_ADDR(error))
            goto out_close;
    }
```

从上面的注释就可看出上面一段代码的作用。用个图说明更易懂。



上面的 padzero()就是来对“未满一页而与初始化部分共享的部分”进行清零。而 do_brk()是映射 zero page 到该 BSS 空间。

这部分也就是为什么，当程序开始执行时，凡是未初始化变量的初始值为零的缘故。Linux 是在 kernel 里面实现的，而 Windows 好像是在 c 库里面实现的。

```
    *interp_load_addr = load_addr;
    error = ((unsigned long)interp_elf_ex->e_entry) + load_addr;
```

*interp_load_addr 指针带回动态链接器模块载入的基址。Error 作为load_elf_interp()的返回值，带回动态链接器执行的入口地址。

setup_arg_pages源码

```
int setup_arg_pages(struct linux_binprm *bprm,
       unsigned long stack_top,
       int executable_stack)
{
unsigned long stack_base;
struct vm_area_struct *mpnt;
struct mm_struct *mm = current->mm;
int i, ret;
long arg_size;

#ifdef CONFIG_STACK_GROWSUP
/* Move the argument and environment strings to the bottom of the
 * stack space.
 */
int offset, j;
char *to, *from;

/* Start by shifting all the pages down */
i = 0;
for (j = 0; j < MAX_ARG_PAGES; j++) {
    struct page *page = bprm->page[j];
    if (!page)
        continue;
    bprm->page[i++] = page;
}

/* Now move them within their pages */
offset = bprm->p % PAGE_SIZE;
to = kmap(bprm->page[0]);
for (j = 1; j < i; j++) {
    memmove(to, to + offset, PAGE_SIZE - offset);
    from = kmap(bprm->page[j]);
    memcpy(to + PAGE_SIZE - offset, from, offset);
    kunmap(bprm->page[j - 1]);
    to = from;
}
memmove(to, to + offset, PAGE_SIZE - offset);
kunmap(bprm->page[j - 1]);

/* Limit stack size to 1GB */
```

```
    stack_base = current->signal->rlim[RLIMIT_STACK].rlim_max;
    if (stack_base > (1 << 30))
        stack_base = 1 << 30;
    stack_base = PAGE_ALIGN(stack_top - stack_base);

    /* Adjust bprm->p to point to the end of the strings. */
    bprm->p = stack_base + PAGE_SIZE * i - offset;

    mm->arg_start = stack_base;
    arg_size = i << PAGE_SHIFT;

    /* zero pages that were copied above */
    while (i < MAX_ARG_PAGES)
        bprm->page[i++] = NULL;
    #else
    stack_base = arch_align_stack(stack_top - MAX_ARG_PAGES*PAGE_SIZE);
    stack_base = PAGE_ALIGN(stack_base);
    bprm->p += stack_base;
    mm->arg_start = bprm->p;
    arg_size = stack_top - (PAGE_MASK & (unsigned long) mm->arg_start);
    #endif

    arg_size += EXTRA_STACK_VM_PAGES * PAGE_SIZE;

    if (bprm->loader)
        bprm->loader += stack_base;
    bprm->exec += stack_base;

    mpnt = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
    if (!mpnt)
        return -ENOMEM;

    memset(mpnt, 0, sizeof(*mpnt));

    down_write(&mm->mmap_sem);
    {
        mpnt->vm_mm = mm;
    #ifdef CONFIG_STACK_GROWSUP
        mpnt->vm_start = stack_base;
        mpnt->vm_end = stack_base + arg_size;
    #else
        mpnt->vm_end = stack_top;
        mpnt->vm_start = mpnt->vm_end - arg_size;
    #endif
```

```
        /* Adjust stack execute permissions; explicitly enable
         * for EXSTACK_ENABLE_X, disable for EXSTACK_DISABLE_X
         * and leave alone (arch default) otherwise. */
        if (unlikely(executable_stack == EXSTACK_ENABLE_X))
            mpnt->vm_flags = VM_STACK_FLAGS |  VM_EXEC;
        else if (executable_stack == EXSTACK_DISABLE_X)
            mpnt->vm_flags = VM_STACK_FLAGS & ~VM_EXEC;
        else
            mpnt->vm_flags = VM_STACK_FLAGS;
        mpnt->vm_flags |= mm->def_flags;
        mpnt->vm_page_prot = protection_map[mpnt->vm_flags & 0x7];
        if ((ret = insert_vm_struct(mm, mpnt))) {
            up_write(&mm->mmap_sem);
            kmem_cache_free(vm_area_cachep, mpnt);
            return ret;
        }
        mm->stack_vm = mm->total_vm = vma_pages(mpnt);
    }

    for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
        struct page *page = bprm->page[i];
        if (page) {
            bprm->page[i] = NULL;
            install_arg_page(mpnt, page, stack_base);
        }
        stack_base += PAGE_SIZE;
    }
    up_write(&mm->mmap_sem);

    return 0;
    }
```

setup_arg_pages注释

该函数的参数解释如下:
bprm 是可执行的文件结构，不多介绍。
stack_top 是该 process 的用户栈的顶。
Executable_stack 是一个标志值，有下列 3 个值:

```
/* Stack area protections */
#define EXSTACK_DEFAULT   0 /* Whatever the arch defaults to */
#define EXSTACK_DISABLE_X 1/* Disable executable stacks */
#define EXSTACK_ENABLE_X  2 /* Enable executable stacks */
```

0 表示 stack 是否可执行，不做特殊处理，由所运行的 CPU 决定;
1 表示 stack 的不可执行;
2 表示 stack 的可执行。

```
int setup_arg_pages(struct linux_binprm *bprm,
       unsigned long stack_top,
       int executable_stack)
{
unsigned long stack_base;
struct vm_area_struct *mpnt;
struct mm_struct *mm = current->mm;
int i, ret;
long arg_size;

#ifdef CONFIG_STACK_GROWSUP
```

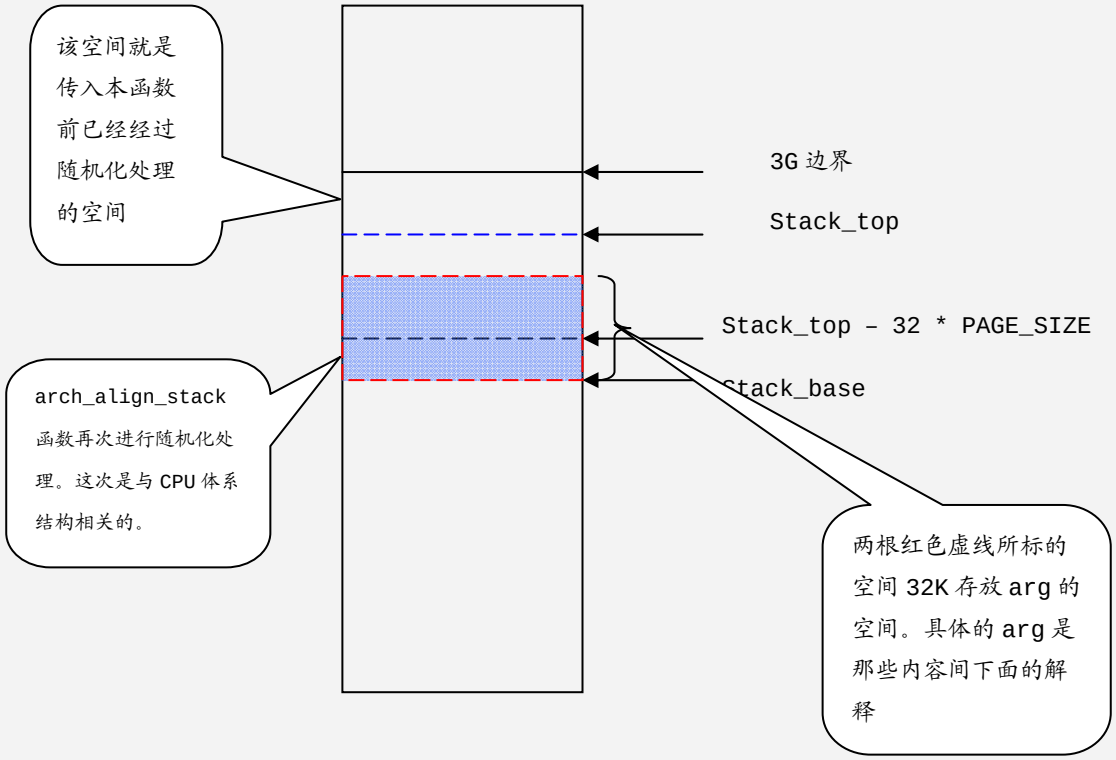
这里是 stack grow up 的方式，不讨论。常规情况下 stack 是 grow down(向下生长的)，所以这按常规逻辑来。除非你在 config 内核时打开 CONFIG_STACK_GROWSUP 选项。

```
/* Move the argument and environment strings to the bottom of the
 * stack space.
 */
int offset, j;
char *to, *from;

/* Start by shifting all the pages down */
i = 0;
for (j = 0; j < MAX_ARG_PAGES; j++) {
    struct page *page = bprm->page[j];
    if (!page)
        continue;
    bprm->page[i++] = page;
}
```

```
    /* Now move them within their pages */
    offset = bprm->p % PAGE_SIZE;
    to = kmap(bprm->page[0]);
    for (j = 1; j < i; j++) {
        memmove(to, to + offset, PAGE_SIZE - offset);
        from = kmap(bprm->page[j]);
        memcpy(to + PAGE_SIZE - offset, from, offset);
        kunmap(bprm->page[j - 1]);
        to = from;
    }
    memmove(to, to + offset, PAGE_SIZE - offset);
    kunmap(bprm->page[j - 1]);

    /* Limit stack size to 1GB */
    stack_base = current->signal->rlim[RLIMIT_STACK].rlim_max;
    if (stack_base > (1 << 30))
        stack_base = 1 << 30;
    stack_base = PAGE_ALIGN(stack_top - stack_base);

    /* Adjust bprm->p to point to the end of the strings. */
    bprm->p = stack_base + PAGE_SIZE * i - offset;

    mm->arg_start = stack_base;
    arg_size = i << PAGE_SHIFT;

    /* zero pages that were copied above */
    while (i < MAX_ARG_PAGES)
        bprm->page[i++] = NULL;
   #else
```

常规方式，即 stack grow down 方式。

```
    stack_base = arch_align_stack(stack_top - MAX_ARG_PAGES*PAGE_SIZE);
```



看个例子:

```
[wzhou@dcmp10 ~]$ ./hello &
[2] 3518
[wzhou@dcmp10 ~]$ Hello, World!


[2]+  Stopped                 ./hello
[wzhou@dcmp10 ~]$ cat /proc/3518/maps
00766000-0077b000 r-xp 00000000 fd:00 492674     /lib/ld-2.3.4.so
0077b000-0077c000 r--p 00014000 fd:00 492674     /lib/ld-2.3.4.so
0077c000-0077d000 rw-p 00015000 fd:00 492674     /lib/ld-2.3.4.so
0077f000-008a2000 r-xp 00000000 fd:00 492914     /lib/tls/libc-2.3.4.so
008a2000-008a3000 r--p 00123000 fd:00 492914     /lib/tls/libc-2.3.4.so
008a3000-008a6000 rw-p 00124000 fd:00 492914     /lib/tls/libc-2.3.4.so
008a6000-008a8000 rw-p 008a6000 00:00 0
08048000-08049000 r-xp 00000000 fd:01 378914     /home/wzhou/hello
08049000-0804a000 rw-p 00000000 fd:01 378914     /home/wzhou/hello
b7fec000-b7fed000 rw-p b7fec000 00:00 0
b7ffe000-b8000000 rw-p b7ffe000 00:00 0
bfeca000-c0000000 rw-p bfeca000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
[wzhou@dcmp10 ~]$
```

上面标红的一行即是现在在讨论的存放 arg 的空间。该空间的首地址 0xbfeca000 是随机的。在 Linux 下用户堆栈的顶在 0xC0000000 - 4，也就是几乎紧挨着用户空间与内核空间的分界线 0xC0000000(3G)。再加上为了避免造成 CPU cache 得过多冲突而空出的一定的随机的空间这里 MAX_ARG_PAGES*PAGE_SIZE 是 32 pages，即 32 * 4K = 128K。这个空间用来存放 3 样东西。

1。可执行文件的文件名。比如/bin/ls

2。可执行文件的参数，也就是 c/c++编程中的 argv[]部分。比如 ls -l -a，就是这里的"-l -a"

3。可执行文件的执行环境。也就是 c/c++编程中的 env[]部分，就是你执行 env command 后的输出。

```
[wzhou@dcmp10 ~]$ env
HOSTNAME=dcmp10
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=::ffff:13.187.241.182 1297 22
SSH_TTY=/dev/pts/9
USER=wzhou
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;01:or=0
1;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm=00;32:*.b
at=00;32:*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:*.taz=00;31:*.lzh=
00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:*.bz=00;31:*.tz=00;31:*.r
pm=00;31:*.cpio=00;31:*.jpg=00;35:*.gif=00;35:*.bmp=00;35:*.xbm=00;35:*.xpm=00;35:*.pn
g=00;35:*.tif=00;35:
KDEDIR=/usr
MAIL=/var/spool/mail/wzhou
PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/wzhou/ald-0.1
.7/bin:/home/wzhou/bin
INPUTRC=/etc/inputrc
PWD=/home/wzhou
LANG=en_US.UTF-8
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SHLVL=1
HOME=/home/wzhou
LOGNAME=wzhou
SSH_CONNECTION=::ffff:13.187.241.182 1297 ::ffff:13.187.243.58 22
LESSOPEN=|/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
_=/bin/env
```

上面三样东西的空间不能超过 128K。

计算出的 stack_base 就是 stack bottom。

```
    stack_base = PAGE_ALIGN(stack_base);
```

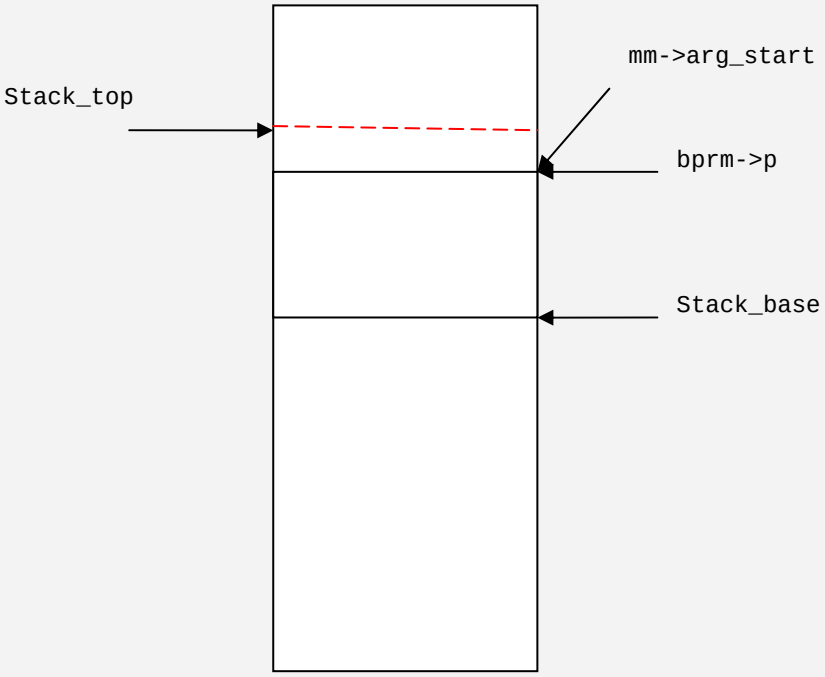
stack bottom 对齐在页边界。

```
    bprm->p += stack_base;
```

在 do_execve()函数的头上有

```
    bprm->p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
```

的赋值，即 bprm->p 指向的是 128K 空间的顶(128K - 4)，现在加上 stack_base，则该 bprm->p 变成了真正的 stack 的 top 地址。




```
    mm->arg_start = bprm->p;
```

arg 的开始即是 bprm->p。

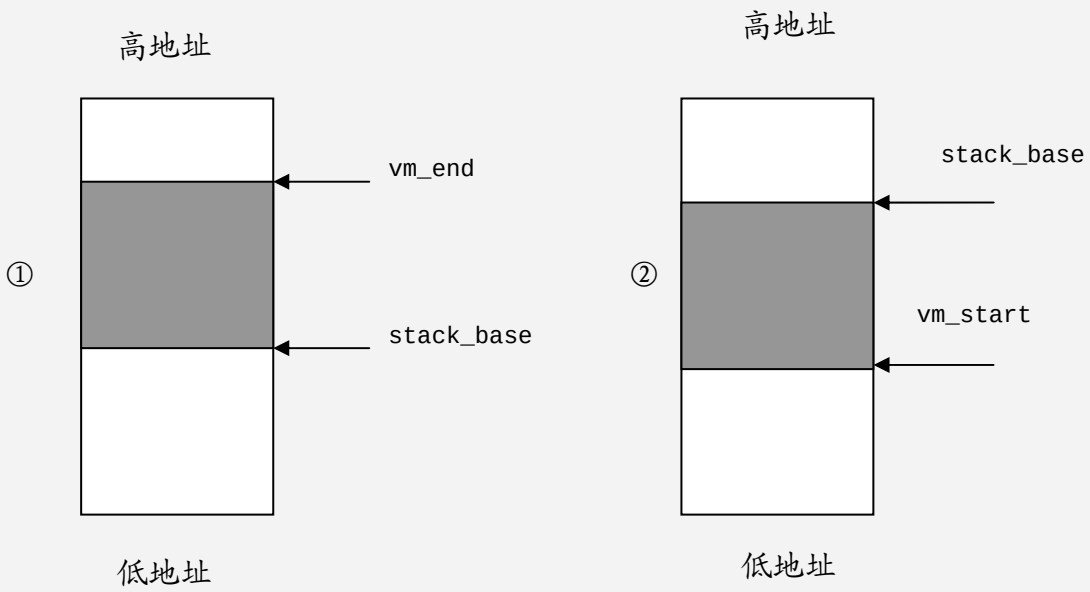
```
    arg_size = stack_top - (PAGE_MASK & (unsigned long) mm->arg_start);
#endif

    arg_size += EXTRA_STACK_VM_PAGES * PAGE_SIZE;
    ???

    if (bprm->loader)
        bprm->loader += stack_base;
    bprm->exec += stack_base;

    mpnt = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
    if (!mpnt)
        return -ENOMEM;
```

虽然存放参数的空间是有了，但还没有被映射进 process 的虚拟地址空间，所以先分配为了管理这一块虚拟地址空间的数据结构 VMA(Virtual Memory Area)。

```
    memset(mpnt, 0, sizeof(*mpnt));
```
初始化该数据结构。

```
    down_write(&mm->mmap_sem);
```
由于要操作当前进程的 memory mapping，所以先获得对 memory mapping 进行保护的 semaphore。

```
    {
        mpnt->vm_mm = mm;
#ifdef CONFIG_STACK_GROWSUP
        mpnt->vm_start = stack_base;
        mpnt->vm_end = stack_base + arg_size;
#else
        mpnt->vm_end = stack_top;
        mpnt->vm_start = mpnt->vm_end - arg_size;
```



上面图①是堆栈向高地址生长的情况，而图②则是堆栈向低地址生长的情况。两个堆栈的 base 正好是相反的。

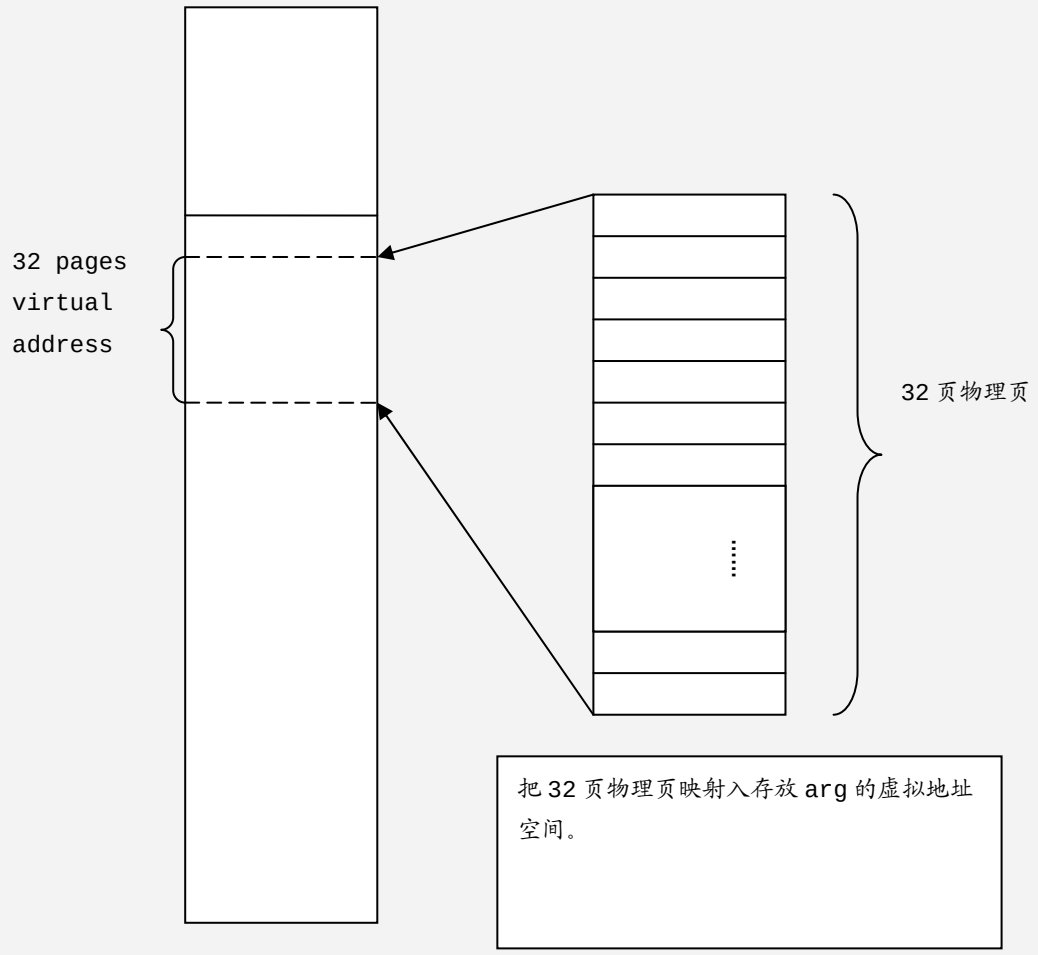
```
#endif
        /* Adjust stack execute permissions; explicitly enable
         * for EXSTACK_ENABLE_X, disable for EXSTACK_DISABLE_X
         * and leave alone (arch default) otherwise. */
        if (unlikely(executable_stack == EXSTACK_ENABLE_X))
            mpnt->vm_flags = VM_STACK_FLAGS |  VM_EXEC;
```
如果 stack 是可执行的，则设置管理该块虚存的 vma 的属性为可执行(VM_EXEC)

```
        else if (executable_stack == EXSTACK_DISABLE_X)
            mpnt->vm_flags = VM_STACK_FLAGS & ~VM_EXEC;
```
如果 stack 是不可执行的，则清除管理该块虚存的 vma 的可执行(VM_EXEC) 属性

```
        else
            mpnt->vm_flags = VM_STACK_FLAGS;
```
但不管怎么样，表示 stack 的 flag(VM_STACK_FLAGS)总要设的。

```
        mpnt->vm_flags |= mm->def_flags;
        mpnt->vm_page_prot = protection_map[mpnt->vm_flags & 0x7];
        if ((ret = insert_vm_struct(mm, mpnt))) {
            up_write(&mm->mmap_sem);
            kmem_cache_free(vm_area_cachep, mpnt);
            return ret;
        }
```
vma 的管理是用红黑树实现的（最起码我几年前看 2.4 内核时是这么实现的，现在应该没变吧，我懒得看了。）所以在这里要把该 vma 的节点要保持"平衡"的插入。

```
        mm->stack_vm = mm->total_vm = vma_pages(mpnt);
    }

    for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
        struct page *page = bprm->page[i];
        if (page) {
            bprm->page[i] = NULL;
            install_arg_page(mpnt, page, stack_base);
        }
        stack_base += PAGE_SIZE;
    }
```
上面的 for 循环为 32 页物理内存建立真正的映射关系，也就是操作 Page Table Entry。这里释放的 page 是在 copy_strings()函数里被申请的，具体请参见该函数的分析。




```
    up_write(&mm->mmap_sem);

    return 0;
}
```

search_binary_handler源码

```
   /*
    * cycle the list of binary formats handler, until one recognizes the
image
    */
   int search_binary_handler(struct linux_binprm *bprm,struct pt_regs
*regs)
   {
   int try,retval;
   struct linux_binfmt *fmt;
   #ifdef __alpha__
   /* handle /sbin/loader.. */
   {
       struct exec * eh = (struct exec *) bprm->buf;

       if (!bprm->loader && eh->fh.f_magic == 0x183 &&
       (eh->fh.f_flags & 0x3000) == 0x3000)
       {
       struct file * file;
       unsigned long loader;

       allow_write_access(bprm->file);
       fput(bprm->file);
       bprm->file = NULL;

           loader = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);

       file = open_exec("/sbin/loader");
       retval = PTR_ERR(file);
       if (IS_ERR(file))
           return retval;

       /* Remember if the application is TASO.  */
       bprm->sh_bang = eh->ah.entry < 0x100000000UL;

       bprm->file = file;
       bprm->loader = loader;
       retval = prepare_binprm(bprm);
       if (retval<0)
           return retval;
       /* should call search_binary_handler recursively here,
```

```
          but it does not matter */
       }
    }
    #endif
    retval = security_bprm_check(bprm);
    if (retval)
        return retval;

    /* kernel module loader fixup */
    /* so we don't try to load run modprobe in kernel space. */
    set_fs(USER_DS);

    retval = audit_bprm(bprm);
    if (retval)
        return retval;

    retval = -ENOENT;
    for (try=0; try<2; try++) {
        read_lock(&binfmt_lock);
        for (fmt = formats ; fmt ; fmt = fmt->next) {
            int  (*fn)(struct  linux_binprm  *,  struct  pt_regs  *)  =
fmt->load_binary;
            if (!fn)
                continue;
            if (!try_module_get(fmt->module))
                continue;
            read_unlock(&binfmt_lock);
            retval = fn(bprm, regs);
            if (retval >= 0) {
                put_binfmt(fmt);
                allow_write_access(bprm->file);
                if (bprm->file)
                    fput(bprm->file);
                bprm->file = NULL;
                current->did_exec = 1;
                proc_exec_connector(current);
                return retval;
            }
            read_lock(&binfmt_lock);
            put_binfmt(fmt);
            if (retval != -ENOEXEC || bprm->mm == NULL)
                break;
            if (!bprm->file) {
                read_unlock(&binfmt_lock);
```

```
            return retval;
         }
      }
      read_unlock(&binfmt_lock);
      if (retval != -ENOEXEC || bprm->mm == NULL) {
         break;
   #ifdef CONFIG_KMOD
      }else{
   #define printable(c) (((c)=='\t') || ((c)=='\n') || (0x20<=(c) &&
(c)<=0x7e))
         if (printable(bprm->buf[0]) &&
            printable(bprm->buf[1]) &&
            printable(bprm->buf[2]) &&
            printable(bprm->buf[3]))
            break; /* -ENOEXEC */
         request_module("binfmt-%04x",       *(unsigned       short
*)(&bprm->buf[2]));
   #endif
      }
   }
   return retval;
   }
```

search_binary_handler注释

本函数用于为特定格式的可执行文件找到其对应的载入器。

```
/*
 * cycle the list of binary formats handler, until one recognizes the image
 */
int  search_binary_handler(struct  linux_binprm  *bprm,struct  pt_regs
*regs)
{
    int try,retval;
    struct linux_binfmt *fmt;
#ifdef __alpha__
    /* handle /sbin/loader.. */
    {
        struct exec * eh = (struct exec *) bprm->buf;

        if (!bprm->loader && eh->fh.f_magic == 0x183 &&
        (eh->fh.f_flags & 0x3000) == 0x3000)
        {
        struct file * file;
        unsigned long loader;

        allow_write_access(bprm->file);
        fput(bprm->file);
        bprm->file = NULL;

            loader = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);

        file = open_exec("/sbin/loader");
        retval = PTR_ERR(file);
        if (IS_ERR(file))
            return retval;

        /* Remember if the application is TASO.  */
        bprm->sh_bang = eh->ah.entry < 0x100000000UL;

        bprm->file = file;
        bprm->loader = loader;
        retval = prepare_binprm(bprm);
        if (retval<0)
            return retval;
        /* should call search_binary_handler recursively here,
           but it does not matter */
```

```
        }
    }
#endif
```

上面的代码都是为 ALPHA CPU(曾经号称世界上最快的 CPU。我记得，最起码在 20 世纪末的时候是这样的)的，忽略。想分析也没办法，我哪来 ALPHA CPU。

```
    retval = security_bprm_check(bprm);
    if (retval)
        return retval;
```

安全性相关，忽略。

```
    /* kernel module loader fixup */
    /* so we don't try to load run modprobe in kernel space. */
    set_fs(USER_DS);
```

我把相关定义抄了过来，

```
#define set_fs(x)   (current_thread_info()->addr_limit = (x))
#define USER_DS     MAKE_MM_SEG(PAGE_OFFSET)
#define MAKE_MM_SEG(s) ((mm_segment_t) { (s) })
```

化简一下，

```
current_thread_info()->addr_limit = 3G
```

也就是设置用户空间的范围。

```
    retval = audit_bprm(bprm);
    if (retval)
        return retval;
```

审计（audit）bprm。与载入没多大关系，忽略。随着内核的发展，零碎的东西越来越多。

```
    retval = -ENOENT;
    for (try=0; try<2; try++) {
```

下面就是去尝试让挂在全局链表头 formats 上的各个可执行文件载入器来辨认并作载入了。这里之所以要对该链表枚举两次，意图如下:
第一遍枚举，如果在链表上没有一个节点认识该可执行文件的格式，则内核还会做一次尝试。在尝试以前，它将根据该可执行文件的 signature 来拼凑出一个可载入内核模块的名字来，然后通过 request_module（）意图载入该 LKM(Loadable Kernel Module)，OS 认为该 LKM 可以向 OS 注册为当前可执行文件的载入器。然后再次对 formats 链表进行枚举，希望这次能够成功。当然几乎可以肯定会失败。但也告诉开发者，完全可以开发一个自己独立的某种可执行文件格式的载入器，而且难度到也不大。难度倒在你怎么定义一个通用灵活强大的可执行文件格式。

```
        read_lock(&binfmt_lock);
```

整个 formats 链表被 binfmt_lock 来同步加锁保护。在枚举列表以前，先锁住链表。

```
        for (fmt = formats ; fmt ; fmt = fmt->next) {
```

对链表上的节点进行枚举

```
            int  (*fn)(struct  linux_binprm  *,  struct  pt_regs  *)  =
fmt->load_binary;
```
取得 load_binary 函数指针。

```
            if (!fn)
                continue;
```
如果为空，则跳过当前节点。

```
            if (!try_module_get(fmt->module))
                continue;
```
增加载入器模块的引用计数，以免在运行时，该模块被卸载。

```
            read_unlock(&binfmt_lock);
```
解锁。
```
            retval = fn(bprm, regs);
```
调用载入器，如果是 ELF 格式可执行文件的话，则调用 load_elf_binary()。

```
            if (retval >= 0) {
```
载入成功。
```
                put_binfmt(fmt);
```
对应上面的 try_module_get(fmt->module)调用，递减载入器模块的引用计数。

```
                allow_write_access(bprm->file);
```
可执行文件正在被载入时，当然是禁止“写”的；载入好以后，允许。这就是 Linux 的特点，正在运行的程序的文件允许被写，允许被删除。

```
                if (bprm->file)
                    fput(bprm->file);
```
递减可执行文件的引用计数，到零时，将关闭。

```
                bprm->file = NULL;
                current->did_exec = 1;
```
设执行标志。
```
                proc_exec_connector(current);
```
看函数名大概与 proc 文件系统有关。忽略。

```
                return retval;
            }
            read_lock(&binfmt_lock);
```
再次锁住 formats 链表。

```
            put_binfmt(fmt);
            if (retval != -ENOEXEC || bprm->mm == NULL)
```

```
                break;
```
如果是这种错误，表示载入器是认识该可执行文件的，但在载入之时，出错了，所以退出。

```
            if (!bprm->file) {
                read_unlock(&binfmt_lock);
                return retval;
            }
        }
        read_unlock(&binfmt_lock);
        if (retval != -ENOEXEC || bprm->mm == NULL) {
            break;
```
运行到这里，表示链表中没有节点认识该可执行文件。

```
#ifdef CONFIG_KMOD
        }else{
#define printable(c) (((c)=='\t') || ((c)=='\n') || (0x20<=(c) &&
(c)<=0x7e))
            if (printable(bprm->buf[0]) &&
                printable(bprm->buf[1]) &&
                printable(bprm->buf[2]) &&
                printable(bprm->buf[3]))
                break; /* -ENOEXEC */
            request_module("binfmt-%04x",        *(unsigned        short
*)(&bprm->buf[2]));
#endif
```
只有在编译内核时打开 LKM 支持的情况下才谈得上动态载入对应内核模块。取出可执行文件签名的头 4 个之母，拚出模块名，要求载入。

request_module()是个很有趣的东东，它是内核中难得的 kernel mode 去请求 user mode 做事的函数。

```
        }
    }
    return retval;
}
```

create_elf_tables源码

```
static int
create_elf_tables(struct linux_binprm *bprm, struct elfhdr *exec,
    int interp_aout, unsigned long load_addr,
    unsigned long interp_load_addr)
{
unsigned long p = bprm->p;
int argc = bprm->argc;
int envc = bprm->envc;
elf_addr_t __user *argv;
elf_addr_t __user *envp;
elf_addr_t __user *sp;
elf_addr_t __user *u_platform;
const char *k_platform = ELF_PLATFORM;
int items;
elf_addr_t *elf_info;
int ei_index = 0;
struct task_struct *tsk = current;

/*
 * If this architecture has a platform capability string, copy it
 * to userspace.  In some cases (Sparc), this info is impossible
 * for userspace to get any other way, in others (i386) it is
 * merely difficult.
 */
u_platform = NULL;
if (k_platform) {
    size_t len = strlen(k_platform) + 1;

    /*
     * In some cases (e.g. Hyper-Threading), we want to avoid L1
     * evictions by the processes running on the same package. One
     * thing we can do is to shuffle the initial stack for them.
     */

    p = arch_align_stack(p);

    u_platform = (elf_addr_t __user *)STACK_ALLOC(p, len);
    if (__copy_to_user(u_platform, k_platform, len))
        return -EFAULT;
}
```

```
    /* Create the ELF interpreter info */
    elf_info = (elf_addr_t *)current->mm->saved_auxv;
    #define NEW_AUX_ENT(id, val) \
    do { \
        elf_info[ei_index++] = id; \
        elf_info[ei_index++] = val; \
    } while (0)

    #ifdef ARCH_DLINFO
    /*
     * ARCH_DLINFO must come first so PPC can do its special alignment
of
     * AUXV.
     */
    ARCH_DLINFO;
    #endif
    NEW_AUX_ENT(AT_HWCAP, ELF_HWCAP);
    NEW_AUX_ENT(AT_PAGESZ, ELF_EXEC_PAGESIZE);
    NEW_AUX_ENT(AT_CLKTCK, CLOCKS_PER_SEC);
    NEW_AUX_ENT(AT_PHDR, load_addr + exec->e_phoff);
    NEW_AUX_ENT(AT_PHENT, sizeof(struct elf_phdr));
    NEW_AUX_ENT(AT_PHNUM, exec->e_phnum);
    NEW_AUX_ENT(AT_BASE, interp_load_addr);
    NEW_AUX_ENT(AT_FLAGS, 0);
    NEW_AUX_ENT(AT_ENTRY, exec->e_entry);
    NEW_AUX_ENT(AT_UID, tsk->uid);
    NEW_AUX_ENT(AT_EUID, tsk->euid);
    NEW_AUX_ENT(AT_GID, tsk->gid);
    NEW_AUX_ENT(AT_EGID, tsk->egid);
        NEW_AUX_ENT(AT_SECURE, security_bprm_secureexec(bprm));
    if (k_platform) {
        NEW_AUX_ENT(AT_PLATFORM,
                (elf_addr_t)(unsigned long)u_platform);
    }
    if (bprm->interp_flags & BINPRM_FLAGS_EXECFD) {
        NEW_AUX_ENT(AT_EXECFD, bprm->interp_data);
    }
    #undef NEW_AUX_ENT
    /* AT_NULL is zero; clear the rest too */
    memset(&elf_info[ei_index], 0,
          sizeof   current->mm->saved_auxv   -   ei_index   *   sizeof
elf_info[0]);
```

```
    /* And advance past the AT_NULL entry.  */
    ei_index += 2;

    sp = STACK_ADD(p, ei_index);

    items = (argc + 1) + (envc + 1);
    if (interp_aout) {
        items += 3; /* a.out interpreters require argv & envp too */
    } else {
        items += 1; /* ELF interpreters only put argc on the stack */
    }
    bprm->p = STACK_ROUND(sp, items);

    /* Point sp at the lowest address on the stack */
    #ifdef CONFIG_STACK_GROWSUP
    sp = (elf_addr_t __user *)bprm->p - items - ei_index;
    bprm->exec = (unsigned long)sp; /* XXX: PARISC HACK */
    #else
    sp = (elf_addr_t __user *)bprm->p;
    #endif

    /* Now, let's put argc (and argv, envp if appropriate) on the stack
*/
    if (__put_user(argc, sp++))
        return -EFAULT;
    if (interp_aout) {
        argv = sp + 2;
        envp = argv + argc + 1;
        __put_user((elf_addr_t)(unsigned long)argv, sp++);
        __put_user((elf_addr_t)(unsigned long)envp, sp++);
    } else {
        argv = sp;
        envp = argv + argc + 1;
    }

    /* Populate argv and envp */
    p = current->mm->arg_end = current->mm->arg_start;
    while (argc-- > 0) {
        size_t len;
        __put_user((elf_addr_t)p, argv++);
        len = strnlen_user((void __user *)p, PAGE_SIZE*MAX_ARG_PAGES);
        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
            return 0;
        p += len;
```

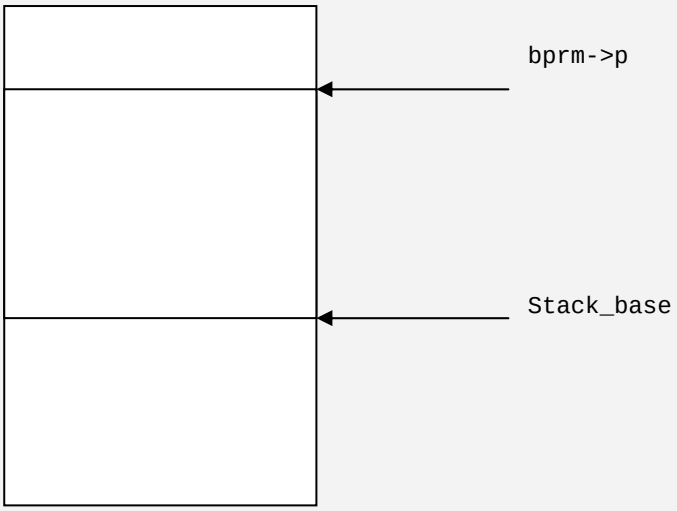
```
    }
    if (__put_user(0, argv))
        return -EFAULT;
    current->mm->arg_end = current->mm->env_start = p;
    while (envc-- > 0) {
        size_t len;
        __put_user((elf_addr_t)p, envp++);
        len = strnlen_user((void __user *)p, PAGE_SIZE*MAX_ARG_PAGES);
        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
            return 0;
        p += len;
    }
    if (__put_user(0, envp))
        return -EFAULT;
    current->mm->env_end = p;

    /* Put the elf_info on the stack in the right place.  */
    sp = (elf_addr_t __user *)envp + 1;
    if (copy_to_user(sp, elf_info, ei_index * sizeof(elf_addr_t)))
        return -EFAULT;
    return 0;
    }
```

create_elf_tables注释

这里的参数解释如下:
bprm 就是当前载入的可执行文件的 linux_binprm 结构
exec 为当前载入的可执行文件的 ELF 头
interp_aout = 1，表示载入的是 aout 型可执行文件；为 0，表示为 ELF 型可执行文件
load_addr 为当前可执行文件的载入基址，一般为 08048000。不知道为什么，Linux 下的 GNU linker 比较偏好于该地址,就象 Windows 下 Microsoft 的 linker 喜欢 4M 边界一样。
interp_load_addr 为动态链接器被载入的基址。
如果用 KDB 调式内核的话，要注意，在 kernel 的反汇编代码中是找不到该函数的 symbol 的，因为该函数被就地展开在 load_elf_binary()中。

```
static int
create_elf_tables(struct linux_binprm *bprm, struct elfhdr *exec,
       int interp_aout, unsigned long load_addr,
       unsigned long interp_load_addr)
{
    unsigned long p = bprm->p;
```

在 create_arg_pages()函数中已经详细介绍过该变量(bprm->p)。令 p 指向存放 arg 参数的 stack 的栈顶(top)。




```
    int argc = bprm->argc;
    int envc = bprm->envc;
    elf_addr_t __user *argv;
    elf_addr_t __user *envp;
    elf_addr_t __user *sp;
    elf_addr_t __user *u_platform;
```

```
    const char *k_platform = ELF_PLATFORM;
```
在我的调试机上是字符串“**i686**”。

```
    int items;
    elf_addr_t *elf_info;
    int ei_index = 0;
    struct task_struct *tsk = current;

    /*
     * If this architecture has a platform capability string, copy it
     * to userspace.  In some cases (Sparc), this info is impossible
     * for userspace to get any other way, in others (i386) it is
     * merely difficult.
     */
    u_platform = NULL;
    if (k_platform) {
        size_t len = strlen(k_platform) + 1;
```
len = strlen("i686") + 1 = 6
```
        /*
         * In some cases (e.g. Hyper-Threading), we want to avoid L1
         * evictions by the processes running on the same package. One
         * thing we can do is to shuffle the initial stack for them.
         */

        p = arch_align_stack(p);
```
从上面的注释看，由于象超线程等的原因，如果进程在同一个地方设置 stack 而造成 L1 cache 的冲突，所以这里要打乱或随机化 stack 的设置。显然该函数与 CPU 的体系结构相关，很多架构的 CPU 对该函数的定义是空。而 Intel CPU 的定义如下:

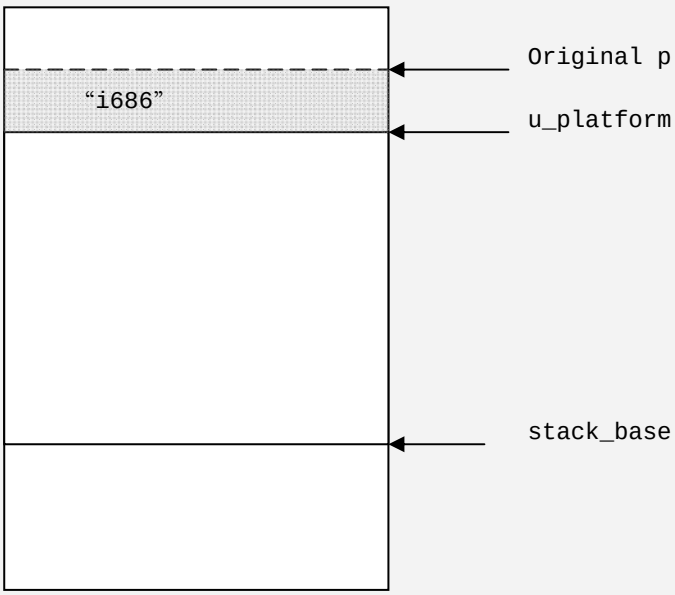
```
unsigned long arch_align_stack(unsigned long sp)
{
    if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
        sp -= get_random_int() % 8192;
    return sp & ~0xf;
}
```
显然是对 stack bottom 指针一定的随机化。

```
        u_platform = (elf_addr_t __user *)STACK_ALLOC(p, len);
```
STACK_ALLOC macro 的定义很有趣。如下:

```
#ifdef CONFIG_STACK_GROWSUP
#define STACK_ALLOC(sp, len) ({ \
    elf_addr_t __user *old_sp = (elf_addr_t __user *)sp; sp += len; \
    old_sp; })
#else
```

```
#define STACK_ALLOC(sp, len) ({ sp -= len ; sp; })
#endif
```

对于不同增长方向的 stack，分配方式是不同的。我们这里当然是最普通的方式：向下增长的 stack。这里在 stack 上留出 len 个 bytes 的空间。




```
        if (__copy_to_user(u_platform, k_platform, len))
            return -EFAULT;
    }
```

把象“i686”这样的字符串从内核空间拷贝到用户空间的 stack 上。

```
    /* Create the ELF interpreter info */
    elf_info = (elf_addr_t *)current->mm->saved_auxv;
#define NEW_AUX_ENT(id, val) \
    do { \
        elf_info[ei_index++] = id; \
        elf_info[ei_index++] = val; \
    } while (0)

#ifdef ARCH_DLINFO
    /*
     * ARCH_DLINFO must come first so PPC can do its special alignment
of
     * AUXV.
     */
    ARCH_DLINFO;
#endif
    NEW_AUX_ENT(AT_HWCAP, ELF_HWCAP);
```

```
    NEW_AUX_ENT(AT_PAGESZ, ELF_EXEC_PAGESIZE);
    NEW_AUX_ENT(AT_CLKTCK, CLOCKS_PER_SEC);
    NEW_AUX_ENT(AT_PHDR, load_addr + exec->e_phoff);
    NEW_AUX_ENT(AT_PHENT, sizeof(struct elf_phdr));
    NEW_AUX_ENT(AT_PHNUM, exec->e_phnum);
    NEW_AUX_ENT(AT_BASE, interp_load_addr);
    NEW_AUX_ENT(AT_FLAGS, 0);
    NEW_AUX_ENT(AT_ENTRY, exec->e_entry);
    NEW_AUX_ENT(AT_UID, tsk->uid);
    NEW_AUX_ENT(AT_EUID, tsk->euid);
    NEW_AUX_ENT(AT_GID, tsk->gid);
    NEW_AUX_ENT(AT_EGID, tsk->egid);
    NEW_AUX_ENT(AT_SECURE, security_bprm_secureexec(bprm));
    if (k_platform) {
        NEW_AUX_ENT(AT_PLATFORM,
               (elf_addr_t)(unsigned long)u_platform);
    }
    if (bprm->interp_flags & BINPRM_FLAGS_EXECFD) {
        NEW_AUX_ENT(AT_EXECFD, bprm->interp_data);
    }
#undef NEW_AUX_ENT
    /* AT_NULL is zero; clear the rest too */
    memset(&elf_info[ei_index], 0,
          sizeof   current->mm->saved_auxv   -   ei_index   *   sizeof
elf_info[0]);

    /* And advance past the AT_NULL entry.  */
    ei_index += 2;
```

上面都是为在 proc 文件系统中显示该进程内存相关辅助信息而设。启动一个程序后你可以在目录/proc/xxxx/auxv 下看到。其中 xxxx 是该进程的 process id。

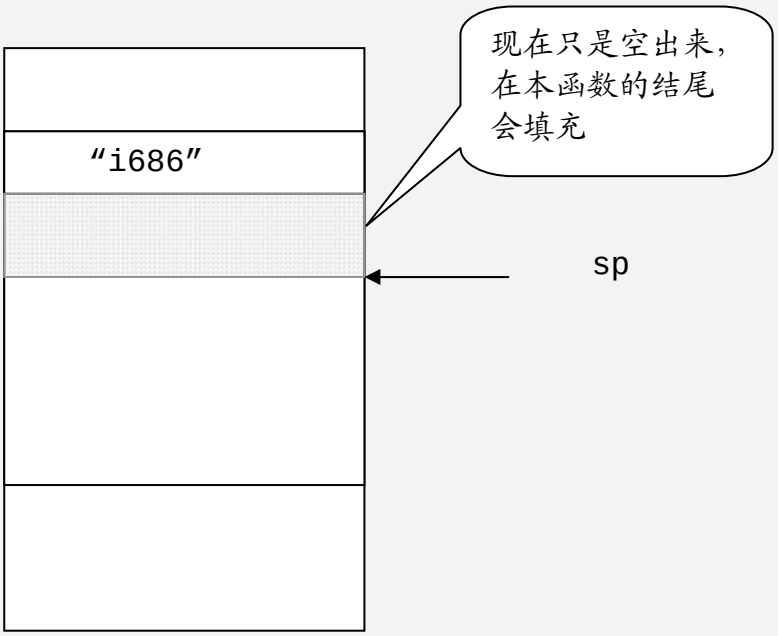
下面摘自 src/linux/auxvec.h 的注释应该能自解释了:

```
/* Symbolic values for the entries in the auxiliary table
   put on the initial stack */
#define AT_NULL   0 /* end of vector */
#define AT_IGNORE 1 /* entry should be ignored */
#define AT_EXECFD 2 /* file descriptor of program */
#define AT_PHDR   3 /* program headers for program */
#define AT_PHENT  4 /* size of program header entry */
#define AT_PHNUM  5 /* number of program headers */
#define AT_PAGESZ 6 /* system page size */
#define AT_BASE   7 /* base address of interpreter */
#define AT_FLAGS  8 /* flags */
#define AT_ENTRY  9 /* entry point of program */
#define AT_NOTELF 10    /* program is not ELF */
```

```
#define AT_UID    11/* real uid */
#define AT_EUID   12    /* effective uid */
#define AT_GID    13/* real gid */
#define AT_EGID   14    /* effective gid */
#define AT_PLATFORM 15  /* string identifying CPU for optimizations */
#define AT_HWCAP  16    /* arch dependent hints at CPU capabilities */
#define AT_CLKTCK 17    /* frequency at which times() increments */

#define AT_SECURE 23   /* secure mode boolean */

#define AT_VECTOR_SIZE  44 /* Size of auxiliary table.  */
```

```
    sp = STACK_ADD(p, ei_index);
```

根据 stack 的增长方向来调整 stack 指针。




```
    items = (argc + 1) + (envc + 1);
```

argc 是程序参数列表 argv[]的个数，而 envc 是环境变量列表 envp[]的个数。分别对它们各加 1，是因为本身在 argv[]与 envp[]中的字符串是一 NULL 结尾的，为了标示整个列表的结尾，它们分别用 NULL NULL 结尾，所以各自多加一个 NULL 字符串。

举例如下:

启动程序: /bin/ls -l -a

则

```
argv[0] = "/bin/ls\0"    (为表示 NULL 结尾，我把 NULL 字符也标示出来)
argv[1] = "-l\0"
argv[2] = "-a\0"
argc = 3
```

```
envp[0] = "HOSTNAME=dcmp10\0"
envp[1] = "TERM=xterm\0"
…
envp[29] = "_=/bin/env\0"
envc = 30
```

当把他们放在内存里时，是这样安排的

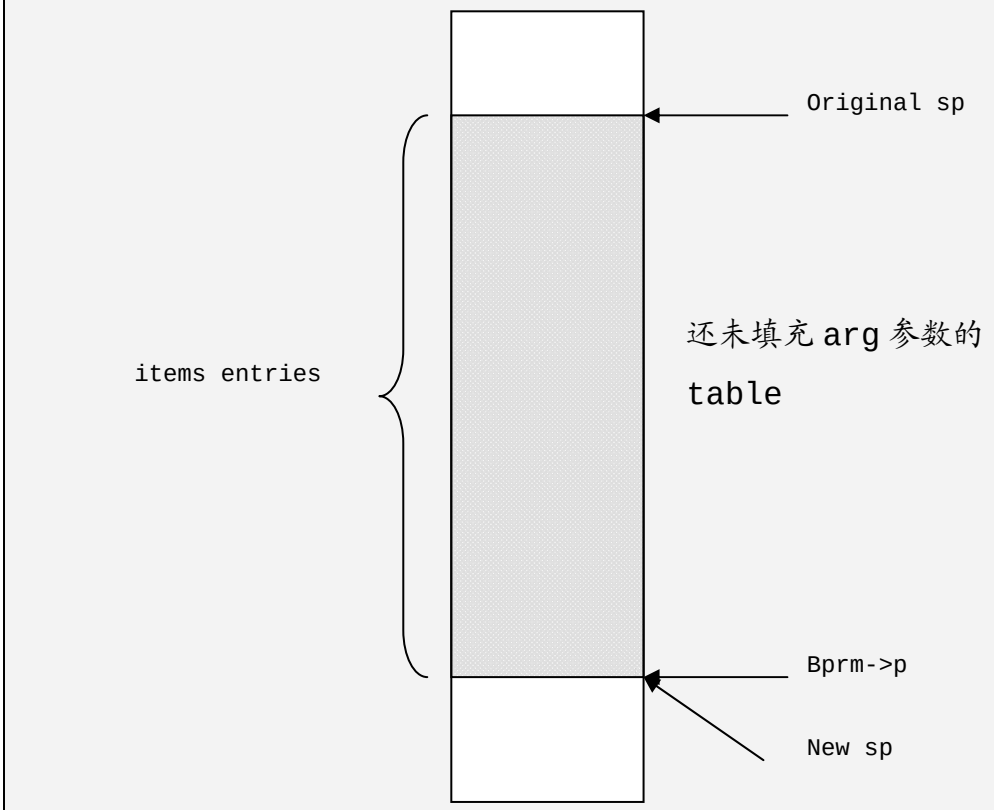
“/bin/ls\0-l\0-a\0\0\0HOSTNAME=dcmp10\0TERM=xterm\0…/bin/env\0\0\0”

注意上面标红的两处“\0\0”。这就是 argc 与 envc 分别要加 1 的原因。用“\0\0”来作为 NULL 字符串数组的结尾。

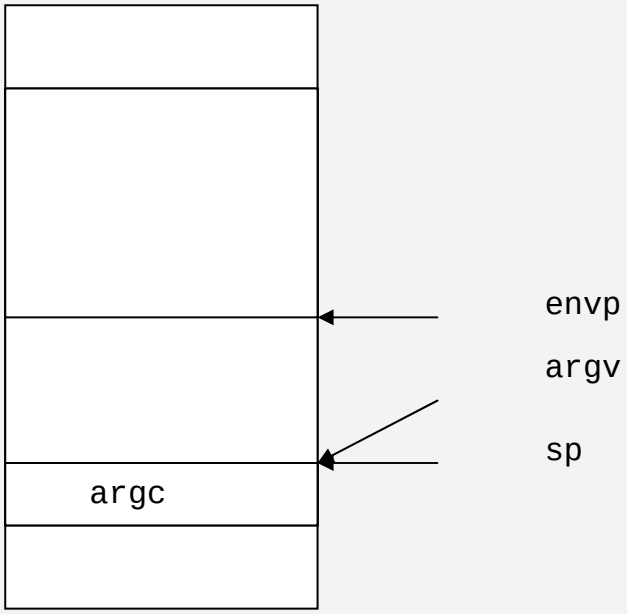
```
    if (interp_aout) {
        items += 3; /* a.out interpreters require argv & envp too */
    } else {
        items += 1; /* ELF interpreters only put argc on the stack */
    }
```

对格式不同，要放入的东西还不一样。

```
    bprm->p = STACK_ROUND(sp, items);

    /* Point sp at the lowest address on the stack */
#ifdef CONFIG_STACK_GROWSUP
    sp = (elf_addr_t __user *)bprm->p - items - ei_index;
    bprm->exec = (unsigned long)sp; /* XXX: PARISC HACK */
#else
    sp = (elf_addr_t __user *)bprm->p;
#endif
```

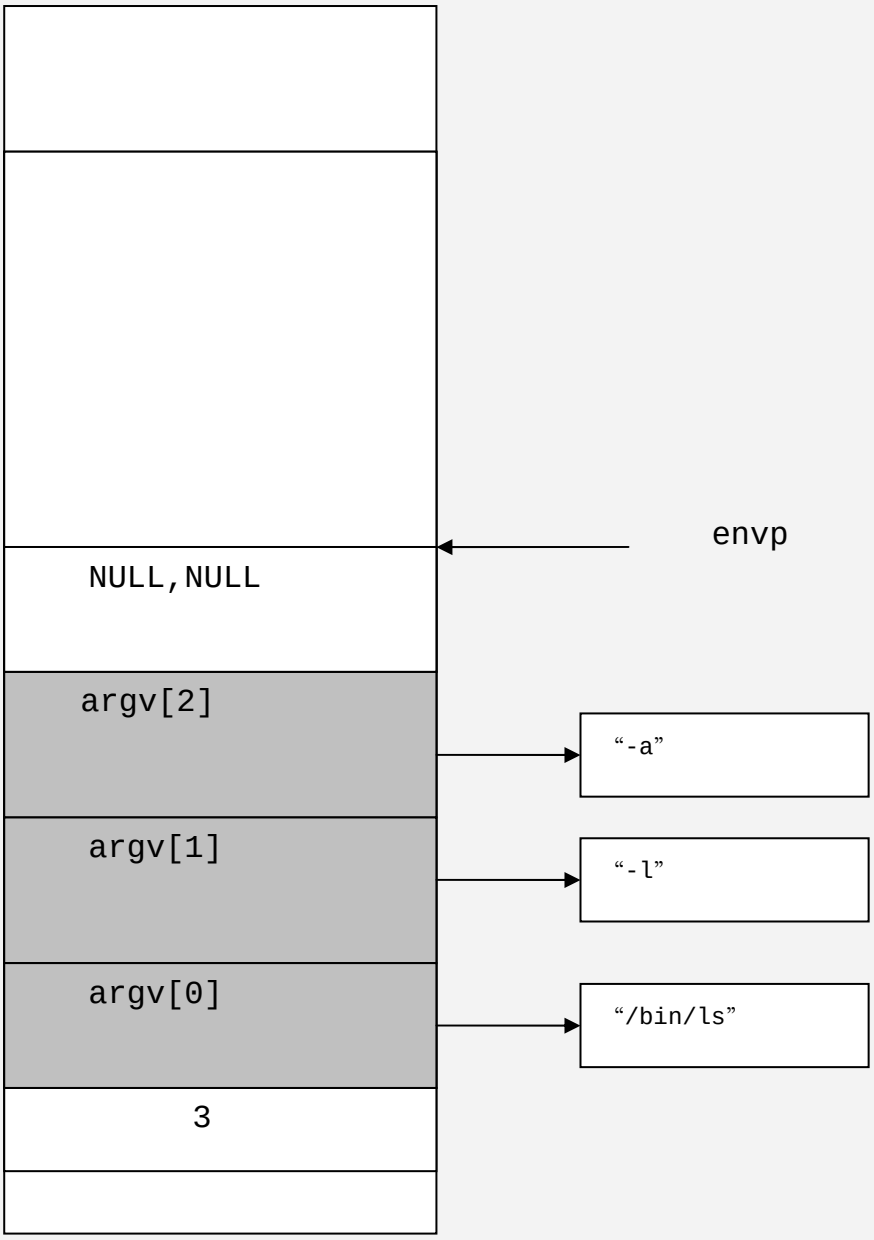



```
    /* Now, let's put argc (and argv, envp if appropriate) on the stack
*/
    if (__put_user(argc, sp++))
        return -EFAULT;
    if (interp_aout) {
        argv = sp + 2;
        envp = argv + argc + 1;
        __put_user((elf_addr_t)(unsigned long)argv, sp++);
        __put_user((elf_addr_t)(unsigned long)envp, sp++);
    } else {
        argv = sp;
        envp = argv + argc + 1;
    }
```



设定 argv 与 envp 指针。

```
    /* Populate argv and envp */
    p = current->mm->arg_end = current->mm->arg_start;
    while (argc-- > 0) {
        size_t len;
        __put_user((elf_addr_t)p, argv++);
        len = strnlen_user((void __user *)p, PAGE_SIZE*MAX_ARG_PAGES);
        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
            return 0;
        p += len;
    }
```



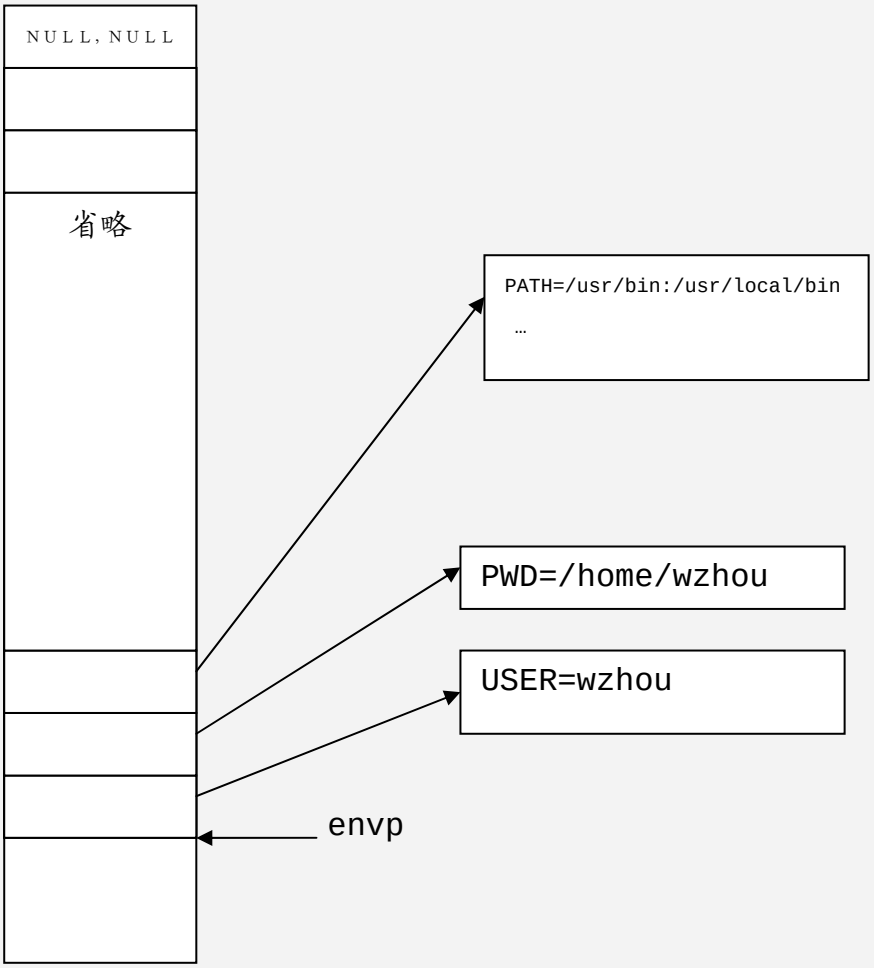
建立 argv[]指针数组，比如对于命令 “/bin/ls –l -a”，则 argc = 3，
argv[0] = “/bin/ls”
argv[1] = “-l”
argv[2] = “-a”

```
    if (__put_user(0, argv))
        return -EFAULT;
```

以两个 NULL 为结尾。

```
    current->mm->arg_end = current->mm->env_start = p;
    while (envc-- > 0) {
        size_t len;
```

```
        __put_user((elf_addr_t)p, envp++);
        len = strnlen_user((void __user *)p, PAGE_SIZE*MAX_ARG_PAGES);
        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
            return 0;
        p += len;
    }
```




```
    if (__put_user(0, envp))
        return -EFAULT;
    current->mm->env_end = p;
```

建立 envp[]数组。

```
    /* Put the elf_info on the stack in the right place.  */
    sp = (elf_addr_t __user *)envp + 1;
    if (copy_to_user(sp, elf_info, ei_index * sizeof(elf_addr_t)))
        return -EFAULT;
    return 0;
}
```

准备 argv[]，envp[]数组。

后记

其实说这是可执行文件的“载入”过程实在有点勉强，因为无论可执行文件本身还是动态链接器，都并没有真正被载入到内存中。到 execute 系统调用完成时，系统只是做好了标记，即虚拟地址空间中的哪一块与可执行文件中的某一段是关联的，内存本身根本没有包含可执行文件的任何东西，甚至连物理内存都没有分配，分配的只是虚拟空间。程序执行的大幕还只是刚刚开始拉开。

这次是重温了大约 4 年前看 2.4.22 内核时的 ELF 可执行文件载入过程，发觉主体代码并没有什么大改变（也确实不应该有大改变，毕竟 ELF Format 的定义是差不多 20 年前的事了）。把以前的笔记整理了一下，遂成此文，倒也没有花很大功夫。

这次看的是当前最新的 kernel --- **2.6.20**。调试环境是建在 Vmware 5.0 for Windows 中的 2.6.20 内核 + KDB (Kernel Debugger)。很久没用 KDB 了，发觉 KDB 增加了许多调试命令，功能更强了，但遗憾的是，稳定性好象不是很好，最起码给我的感觉，没有 2.4.22 时稳定了。

联系

Walter Zhou
z-l-dragon@hotmail.com