基于 exec 函数族分析 Linux 初始化进程运行环境的过程

1 Linux 内存管理简介

Linux 采用了典型的分页式虚拟内存管理机制。我们来看下 Linux 定义的页描述符（Page Descriptor）。

TYPE	NAME	DESCRIPTION
unsigned long	flags	一组标志位的集合
atomic_t	_count	该页框被引用的次数
atomic_t	_mapcount	指向该页框的页表项的数量（-1 表示没有任何页表项指向当前页框）
unsigned long	private	略
struct address_space*	mapping	当页被换入到 page cache 或者该页指向匿名的区域的时候，用来指定页的映射地址（in cache or memory）
unsigned long	index	略
struct list_head	lru	在 Linux 系统中，所有的最近最少使用的页（Least Recently Used, LRU）组成了一个双向链表，这个指针指向这个链表表头

我们关心上述结果的关于标志位 flags 的一些描述，如下表所示

Flags	Meaning
PG_locked	当前页处于锁定状态，例如当前页正在进行 IO 操作
PG_error	当传输当前页时出现了 IO 错误
PG_referenced	该页最近被访问过
PG_dirty	该页被修改过
PG_lru	该页在 active 或者 inactive 链表上
PG_active	该页在 active 链表上

事实上，从上面的一些字段我们就可以粗略地窥探到 Linux 对于 page 的替换策略，整体上应该是基于 LRU 算法，也即最近最少使用算法。Linux 初始化了两个队列，分别是 active 队列和 inactive 队列用于记录最近页的使用情况。算法大致描述如下：

（1）kswapd 守护进程维护着 LRU 队列，这个队列里的元素就是最近一次可能被换出的候选页面；

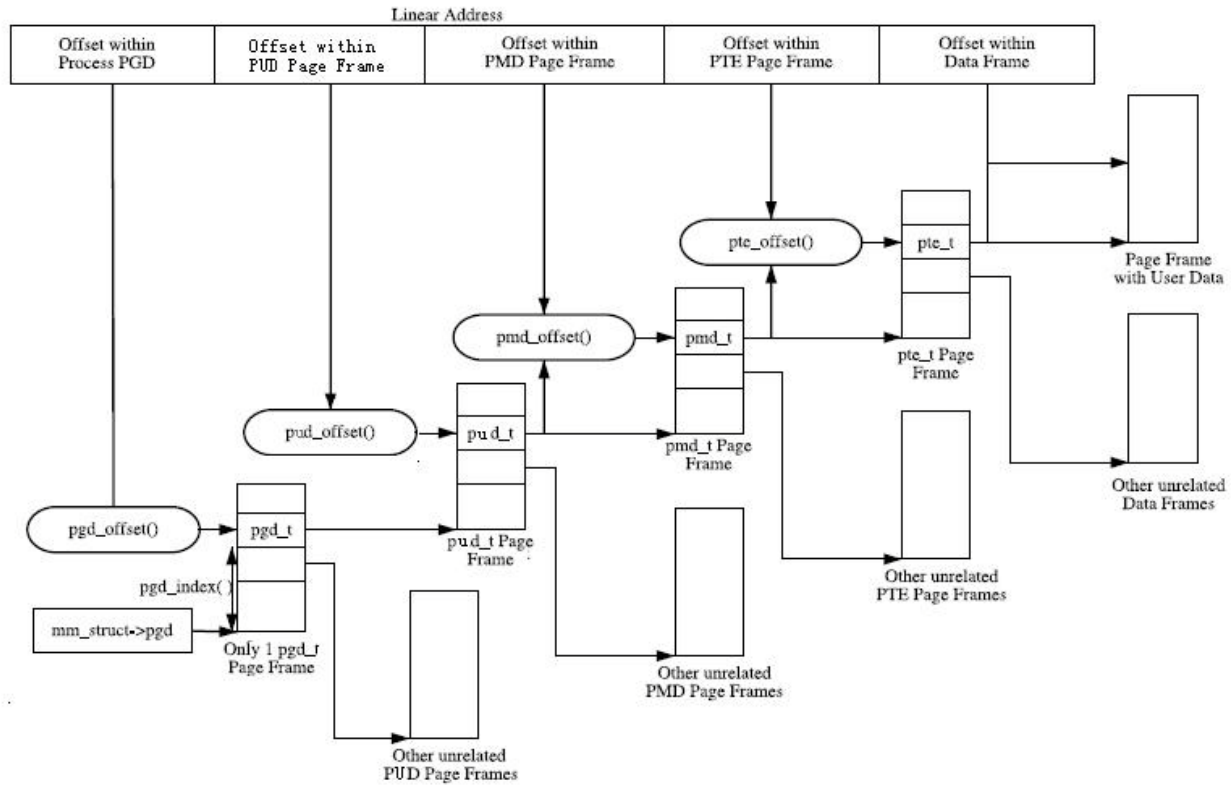
（2）扫描 LRU 队列，将最近既没有被访问过（PG_referenced）的页放入 inactive 队列，否则，将其放入 active 队列；

（3）决定从 inactive 队列中选出应该被替换的页面；

Linux 通过这样两个队列修正了原有的 LRU 算法，同时通过两个队列大大降低误换页面的可能性。

Linux 采用多级分页结果，而在 32 位机子中通常是三级分页 PGD(page global directory) → PMD(page middle directory) → PTE(page table)。为了提供对 64 位机子的支持，在 PGD 和 PMD 之间还有 PUD(page upper directory)。

PGD 每个条目中指向一个 PUD，PUD 的每个条目指向一个 PMD，PMD 的每个条目指向一个 PTE，PTE 的每个条目指向一个页面(Page)的物理首地址。因此一个线性地址被分为了 5 个部分，如下图：



2 进程的内存空间

2.1 进程空间的管理

进程的内存空间在 Linux 内核的管理下被规划成一个个区域，这些区域代表着该进程所能使用的内存，并且这段内存不应该被其他的进程非常占用（除非两个进程共享了同一片内存）。每个内存区域的大小都是 4096（4kb）的整数倍，这样这些内存区域能够完整地占据所分配到的页（每个页的大小默认是 4kb），从而可以提高系统分配和管理这些内存区域的效率。

所有和进程地址空间相关的信息都封装在一个成为内存描述符（*memory descriptor*）的数据结构中，Linux 内核源码中定义为 *mm_struct*。在前面对进程描述符的分析中，我们提到了 *mm* 这个字段，*mm* 就是指向了这里所说的 *mm_struct*，也就是该进程的内存描述符。下面给出 *mm_struct* 的一些重要的字段。

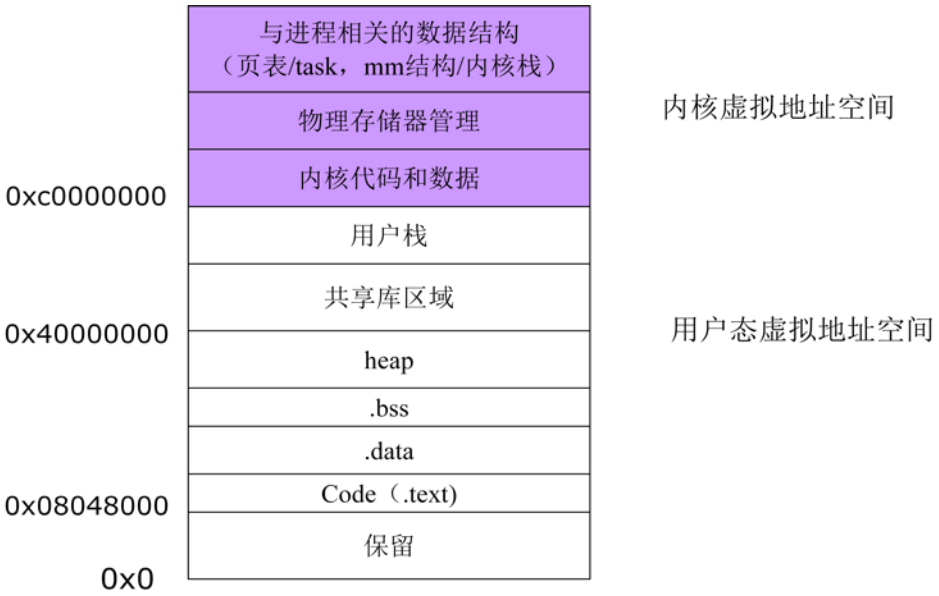
TYPE	NAME	DESCRIPTION
struct	mmap	指向内存区域对象（vm_area_struct）列表表头
pgd_t*	pgd	指向页目录（PGD）
atomic_t	mm_count	记录该内存使用情况的主计数器
atomic_t	mm_users	记录该内存使用情况的从计数器
int	map_count	该进程内存所占用的内存区域的数量

剩下的字段中还包含了对于该进程代码段、数据段、堆栈、命令行参数（arg）和环境变量的起始地址和终止地址。内存描述符在系统中被组织成了一个双向链表 *mmlist*，这个链表的首元素明显是 0 号进程，或者 *init* 进程在系统启动初始化映射的。

上述字段中 *mm_count* 和 *mm_users* 是两个比较重要的字段，因为这两个字段决定了该进程空间是否还存活（仍旧被使用）。*mm_users* 记录了使用这片内存空间的所有轻量级进程的数量，而从同一个父进程 fork 出来的轻量级子进程在 *mm_count* 中算一个单元。比如，某个进程空间由两个拥有同样父进程的轻量级进程共享，那么其 *mm_users* 的值为 2，而 *mm_count* 的值为 1。每次当 *mm_count* 的值递减的时候，系统都会检测其是否变为 0，如果为 0，说明该进程空间已经不被使用，这将导致该内存描述符（进程空间）被回收。如果内核借用了该进程

空间，那么内核会增加 *mm_count*，保证当共享该进程空间的两个进程退出后，*mm_count* 不会变为 0 从而导致进程空间被系统回收。

对于 Linux 内核进程而言，通常它们不需要使用常规内存空间，而它们的内存被映射到高于 *TASK_SIZE* 的地址上（通常是 0xc0000000），下图表示出了 Linux 中常规进程和内核进程对内存的映射情况。



2.2 进程内存区域的组织

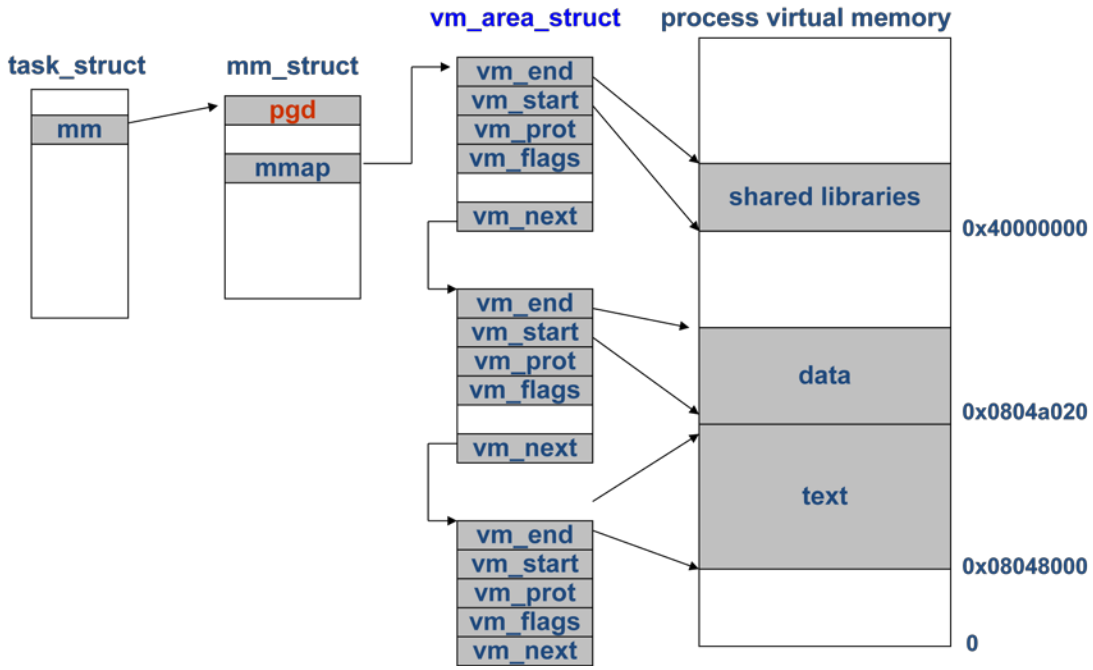
我们知道，进程空间会被划分为一个个内存区域（Memory Regions），那么 Linux 是怎样组织这些内存区域的呢？事实上，Linux 是通过 *vm_area_struct* 管理内存区域，并将这些 *vm_area_struct* 组织成链表的形式。

下表列出了 vm_area_struct 结构中的主要字段。

TYPE	NAME	DESCRIPTION
struct mm_struct*	vm_mm	反向地指向拥有该区域的进程空间的内存描述符
unsigned long	vm_start	该内存区域的起始地址
unsigned long	vm_end	该内存区域的终止地址
struct	vm_next	指向当前进程空间的下一片内存区域
pgprot_t	vm_page_prot	该区域对应页框的访问权限
unsigned long	vm_flags	该区域的标志位，表示该内存区域是和其他进程共享（0）或者是当前进程独有的（1）
struct rb_node	vm_rb	该区域对应的红黑树的节点（用于快速定位该区域）

到这里，和进程空间相关的数据结构基本上已经理清，我们来看看从进程描

述符（*task_struct*）开始，一直到内存区域对象（*vm_area_struct*）在系统中整体的组织关系。如下图所示：



3 基于 exec 分析系统初始化进程运行环境的过程

execve 系统调用可以调用一个可执行文件完全代替当前的进程,它在 libc 中的封装有几个 API:

```
int execl(const charp a t* h n a m e, const char a* rg 0, ... /* (char *) 0 */);
int execv(const charp a t* h n a m e, char *consta rgv [] );
int execle(const charp a t* h n a m e, const char a* rg 0, ...
/* (char *)0, char *cones nt v p [] */);
int execve(const charp a t* h n a m e, char *consta rgv [], char *consten vp [] );
int execlp(const charf i l e* n a m e, const char a* rg 0, ... /* (char *) 0 */);
int execvp(const charf i l e* n a m e, char *consta rgv [] );
```

这几个函数的功能相同，只是参数有所不同，在内核中，它们拥有统一的函数入口 sys_execve。系统为进程运行初始化环境，无非就是完成内存分配和映射以及参数和数据段、代码段和 bss 等的载入。为了对内存分配做进一步了解，我们先看下为进程分配内存空间所调用的 do_mmap 函数。

3.1 do_mmap

```
static inline unsigned long do_mmap(struct file *file, unsigned long addr,
        unsigned long len, unsigned long prot,
        unsigned long flag, unsigned long offset)
{
        unsigned long ret = -EINVAL;
        if ((offset + PAGE_ALIGN(len)) < offset)
                goto out;
        if (!(offset & ~PAGE_MASK))
                ret = do_mmap_pgoff(file, addr, len, prot, flag, offset >> PAGE_SHIFT);
out:
        return ret;
}
```

do_mmap 为当前进程创建和初始化一个新的内存区域。do_mmap 首先对 offset 的值进行一些初始的检测，然后会调用 do_mmap_pgoff 继续执行后面的流程，从而，创建和初始化的工作都在 do_mmap_pgoff 里面完成；do_mmap_pgoff 的代码很长，我们将代码分割进行分析进行分析。

```
if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
    if (!(file && (file->f_vfsmnt->mnt_flags & MNT_NOEXEC)))
        prot |= PROT_EXEC;

if (!len)
    return addr;

/* Careful about overflows.. */
len = PAGE_ALIGN(len);
if (!len || len > TASK_SIZE)
    return -EINVAL;

/* offset overflow? */
if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
    return -EINVAL;

/* Too many mappings? */
if (mm->map_count > sysctl_max_map_count)
    return -ENOMEM;
```

1、这段代码紧跟在参数初始化后面，主要完成对于函数参数的一些检验，以保证这些参数不会出现以下几种情况：

（1）分配的空间大小（*len* 所指定）为 0 或者超过 *TASK_SIZE*（内核空间界限）；

（2）当前进程已经映射了太多内存空间，从而其 mm 所指定的 *map_count* 已经超过系统所允许的最大值；

```
addr = get_unmapped_area(file, addr, len, pgoff, flags);
if (addr & ~PAGE_MASK)
    return addr;

/* Do simple checking here so the lower-level routines won't have
 * to. we assume access permissions have been handled by the open
 * of the memory object, so we don't do any here.
 */
vm_flags = calc_vm_prot_bits(prot) | calc_vm_flag_bits(flags) |
        mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;

if (flags & MAP_LOCKED) {
    if (!can_do_mlock())
        return -EPERM;
    vm_flags |= VM_LOCKED;
}
/* mlock MCL_FUTURE? */
if (vm_flags & VM_LOCKED) {
    unsigned long locked, lock_limit;
    locked = mm->locked_vm << PAGE_SHIFT;
    lock_limit = current->signal->rlim[RLIMIT_MEMLOCK].rlim_cur;
    locked += len;
    if (locked > lock_limit && !capable(CAP_IPC_LOCK))
        return -EAGAIN;
}
```

2、进行一系列的参数检验以后，会继续调用 *get_unmmaped_area* 从系统进程空间中寻找空闲的线性空间。随后是针对权限标记位的设置工作，事实上，就是用函数 flags 参数所传递的权限标志对 *vm_area_struct* 下的 vm_flags 进行赋值。这里我们跳过一段对于文件的操作，因为假定这个进程不会加载任何的文件。

```
munmap_back:
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
    if (vma && vma->vm_start < addr + len) {
        if (do_munmap(mm, addr, len))
            return -ENOMEM;
        goto munmap_back;
    }
```

3、我们知道所有的 vm_area 结构在系统中组成一个链表（为了定位方便还组织成了红黑树），这段函数调用 *find_vma_prepare* 遍历这个链表，直到找到将要插入这个新的地址空间在链表中的直接前驱节点（当然也包括其对应的红黑树的节点）。如果当前新的地址所占用的空间太大了，对于其他地址区域造成了混

叠，那么会调用 *do_munmap* 将当前申请的地址释放掉。

```
/*
 * Can we just expand an old private anonymous mapping?
 * The VM_SHARED test is necessary because shmem_zero_setup
 * will create the file object for a shared anonymous map below.
 */
if (!file && !(vm_flags & VM_SHARED) &&
    vma_merge(mm, prev, addr, addr + len, vm_flags,
                NULL, NULL, pgoff, NULL))
    goto out;
```

4、如果当前新申请的空间为该进程所私有（*VM_SHARED* 没有被设置），并且没有加载文件，这个时候便会调用 *vma_merge* 试图将当前的新的内存区域合并入其直接前驱节点所指定的内存区域中，当然这两个内存区域必须具有相同的访问权限。如果合并成功，那么直接跳出函数，否则继续后面的流程。

```
/*
 * Determine the object being mapped and call the appropriate
 * specific mapper. the address has already been validated, but
 * not unmapped, but the maps are removed from the list.
 */
vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma) {
    error = -ENOMEM;
    goto unacct_error;
}
memset(vma, 0, sizeof(*vma));

vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = protection_map[vm_flags & 0x0f];
vma->vm_pgoff = pgoff;
```

5、这里调用 kmen_cache_alloc 来为这段新的内存地址分配 *vma_area_struct* 结构，并于此同时初始化这个 *vm_area*。

```
if (!file || !vma_merge(mm, prev, addr, vma->vm_end,
        vma->vm_flags, NULL, file, pgoff, vma_policy(vma))) {
    file = vma->vm_file;
    vma_link(mm, vma, prev, rb_link, rb_parent);
    if (correct_wcount)
        atomic_inc(&inode->i_writecount);
```

6、随后调用 vm_link 将新创建的 vma_area_struct 插入到 vma_list 以及对应

的红黑树中。接下来就是一系列的善后工作了。

3.2 do_execve

do_execve 的函数源代码位于./fs/exec.c 文件中，当用户使用 exec 函数族的时候，系统统一调用 do_execve 来完成新的可执行（elf）文件运行前的准备工作，包括内存的分配、数据参数的载入以及对调用 exec 的旧进程的回收。

3.2.1 相关的数据结构

我们先来看下两个与可执行文件相关的数据结构，分别是 *linux_binprm* 和 *linux_binfmt*。内核中为可执行程序的装入定义了一个数据结构 *linux_binprm*，以便将运行一个可执行文件时所需的信息组织在一起。

```
/*
 * This structure is used to hold the arguments that are used when loading binaries.
 */
struct linux_binprm{
    char buf[BINPRM_BUF_SIZE];
    struct page *page[MAX_ARG_PAGES];
    struct mm_struct *mm;
    unsigned long p; /* current top of mem */
    int sh_bang;
    struct file * file;
    int e_uid, e_gid;
    kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
    void *security;
    int argc, envc;
    char * filename;    /* Name of binary as seen by procps */
    char * interp;      /* Name of the binary really executed. Most
                           of the time same as filename, but could be
                           different for binfmt_{misc,script} */
    unsigned interp_flags;
    unsigned interp_data;
    unsigned long loader, exec;
};
```

TYPE	NAME	DESCRIPTION
char	buf	128 个字节的缓存，用于保存可执行文件的文件头（限制大小为 128bytes）
struct page*	page	用于存储参数的页结构，在 Linux 中，进程的每个参数最大使用一个物理页来存储，对于所有的参数而言，最大的存储空间为 32 个页面

struct mm_struct	mm	内核空间，用于暂时存储新进程的可执行文件名、环境变量和新进程的函数变量
unsigned long	p	当前内存的起始地址
int	e_uid, e_gid	分别对应用户 id 和组 id
int	argc, envc	分别对应于参数数量和环境变量数量
char*	filename	Procps（系统进程监视器）所见的二进制文件的名称

在 linux 内核，用 linux_binfmt 结构来表示每一个加载模块，这个结构在系统中组成了一个链表结构。

```
struct linux_binfmt {
    struct linux_binfmt * next;
    struct module *module;
    int (*load_binary)(struct linux_binprm *, struct  pt_regs * regs);
    int (*load_shlib)(struct file *);
    int (*core_dump)(long signr, struct pt_regs * regs, struct file * file);
    unsigned long min_coredump; /* minimal dump size */
};
```

TYPE	NAME	DESCRIPTION
struct linux_binfmt *	next	指向链表下一个元素的指针
struct module *	module	定义该函数所属的模块（module）
函数指针	load_binary	用于加载可执行文件
函数指针	load_shlib	用于加载共享库

在使用这个数据结构前必须调用 *void binfmt_setup* 函数进行初始化；这个函数分别初始化了一些可执行的文件格式，如：*init_elf_binfmt*；*init_aout_binfmt*；*init_java_binfmt*；*init_script_binfmt*。

其实初始化就是用 register_binfmt(struct linux_binfmt * fmt)函数把文件格式注册到系统中，即加入*formats 所指的链中，*formats 的定义如下：

*static struct linux_binfmt *formats = (struct linux_binfmt *) NULL*

在使用装入函数的指针时，如果可执行文件是 ELF 格式的，则指针指向的装入函数分别是：

*load_elf_binary(struct linux_binprm * bprm, struct pt_regs * regs);*

static int load_elf_library(int fd);

do_execve 对可执行文件的载入分成两个阶段，第一个阶段是准备阶段，准备阶段完成对进程参数的预读（读入内核空间）和对于可执行文件格式的判断，并为对应格式的可执行文件选取加载器。第二个阶段就是载入阶段，完成对新进

程数据段、代码段、bss 等等信息向内存的载入。

3.2.2 do_execve 准备阶段

```
retval = -ENOMEM;
bprm = kmalloc(sizeof(*bprm), GFP_KERNEL);
if (!bprm)
    goto out_ret;
memset(bprm, 0, sizeof(*bprm));

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

1、首先调用 *kmalloc* 初始化 *bprm*（*linux_binprm*），随即在内核中打开 filename 所指定的这个可执行文件。

```
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;
```

2、上面这个函数片段主要完成对 bprm 进行初始化和相应字段的赋值。*mm_alloc* 完成对 *bprm*→mm 的分配（内部调用 *do_mmap*，参考前面对这个函数逻辑的相关介绍）和初始化。另外，count 用于计算参数的数量，这里分别计算出了进程

参数的数量并赋值给 *bprm*→argc 和环境变量的数量并赋值给 *bprm*→envc。*prepare_binprm* 执行完以后，可执行文件的文件头（前 128 个字节）就被拷贝到 *bprm*→buf 中。

```
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);
```

3、这段代码调用了三次 copy_strings 的操作。copy_strings 做的就是将参数从调用 execve 的进程先拷贝到 bprm 的 page 中，具体实现就不赘述了。那么为何要经这么一二转手呢？进程调用完 exec 不还是这个进程吗？为何还要拷贝呢？其实虽然还是这个进程，但是其地址空间却重新设置了，原来的地址空间被他 release 了。既然地址空间改变了，那么就必须把新地址空间要用到的东西先转移到一个地方，然后新的地址空间加载以后再把它从这个地方拷贝到新地址空间，那么拷贝到哪里呢？当然是内核了。

4、完成以上对于 *bprm* 的一系列初始化和赋值之后，内核就要开始做正事了，也就是把可执行文件加载到内存中，这个时候系统会调用 *search_binary_handler* 来寻找可执行文件加载模块。我们之前探讨过 linux_binfmt 这个结构，这些模块会被载入成为 linux_binfmt_list 中的一个元素，我们跟进 *search_binary_handler* 来看看内核是怎么完成这个操作的。

search_binary_handler 中有一段代码是专门针对 alpha 处理器的条件编译，这里分析的时候跳过这一段。

```
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;
            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
        }
    }
```

我们知道 *linux_binfmt* 结构在系统中组成了一个链表，那么静态变量 *formats* 便是这个链表的链表头(关于 *formats* 的定义在之前分析 *linux_binfmt* 已经给出)，挂在这个队列中的成员代表着各种可执行文件格式。在 *do_exec* 函数的准备阶段，已经从可执行文件头部读入 128 字节存放在 bprm 的缓冲区中，而且运行所需的参数和环境变量也已收集在 bprm 中。*search_binary_handler* 函数就是逐个扫描 formats 队列，直到找到一个匹配的可执行文件格式，运行的事就交给它。如果在这个队列中没有找到相应的可执行文件格式，就要根据文件头部的信息来查找

是否有为此种格式设计的可动态安装的模块，如果有，就把这个模块安装进内核，并挂入 *formats* 队列，然后再重新扫描。下面给出对上面代码片段的分析：

（1）程序中有两层嵌套 for 循环。内层是针对 *formats* 队列的每个成员，让每一个成员都去执行一下 *load_binary* 函数，如果执行成功，*load_binary* 就把目标文件装入并投入运行，并返回一个正数或 0。当 CPU 从系统调用 *execve* 返回到用户程序时，该目标文件的执行就真正开始了，也就是，子进程新的主体真正开始执行了。如果 *load_binary* 返回一个负数，就说明或者在处理的过程中出错，或者没有找到相应的可执行文件格式，在后一种情况下，返回-ENOEXEC。

（2） 内层循环结束后，如果 *load_binary* 执行失败后的返回值为-ENOEXEC，就说明队列中所有成员都不认识目标文件的格式。这时，如果内核支持动态安装模块（取决于编译选项 *CONFIG_KMOD*），就根据目标文件的第 2 和第 3 个字节生成一个 binfmt 模块，通过 *request_module* 试着将相应的模块装入内核。外层的 for 循环有两次，就是为了在安装了模块以后再来试一次。

（3）*在 linux_binfmt* 数据结构中，有三个函数指针：*load_binary*、*load_shlib* 以及 *core_dump*，其中 *load_binary* 就是具体的装载程序。不同的可执行文件其装载函数也不同，如 *a.out* 格式的装载函数为 *load_aout_binary*，*elf* 的装载函数为 *load_elf_binary*，其源代码分别在 fs/binfmt_aout.c 中和 fs/binfmt_elf 中。

3.2.3 do_execve 载入阶段

由于可执行文件的格式众多，我们选取 elf 文件的载入作为本节的示例。先来看下 *linux_binfmt* 对 elf 格式文件的定义：

```
static struct linux_binfmt elf_format = {
        .module     = THIS_MODULE,
        .load_binary    = load_elf_binary,
        .load_shlib = load_elf_library,
        .core_dump  = elf_core_dump,
        .min_coredump   = ELF_EXEC_PAGESIZE
};
```

下面，我们还是通过内核代码片段完成对 *load_elf_binary* 的分析。

```
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;
```

1、首先，从准备阶段 *bprm* 中的 128 字节缓存中读取 elf 文件头，并同时对文件头进行一些常规检查。当然如果这个可执行文件没有任何执行入口（没有定义 *f_op*）或者没有文件的执行分配任何的空间，那么这个可执行文件的载入只好被中断。

```
/* 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 = (struct elf_phdr *) 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;
}
```

2、这段代码完成对 elf 头信息的检查和读取，关于 *elfhdr* 所定义 elf 头信息的相关介绍可以参考[7]，这里不做深入阐述。

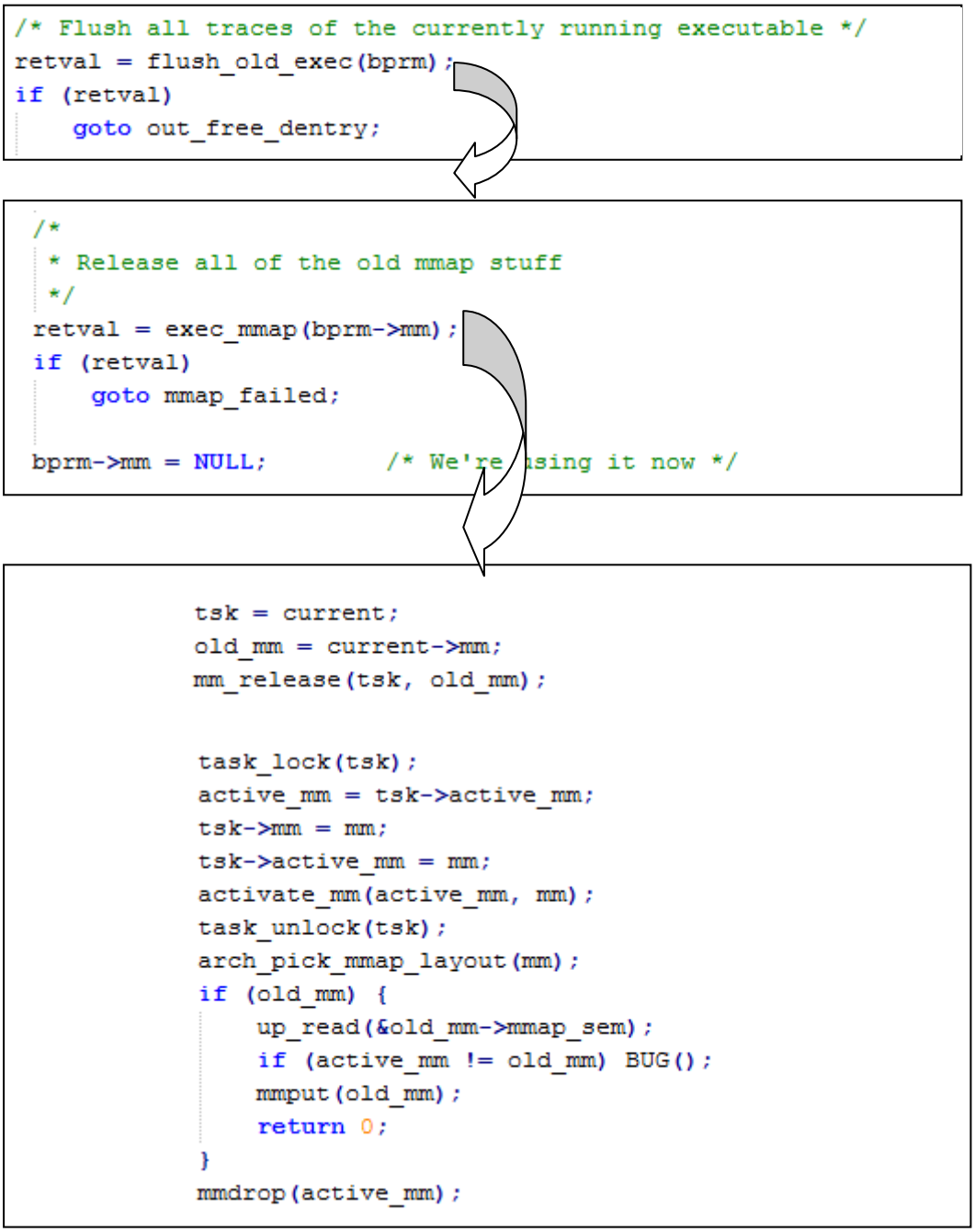
```
/* exec will make our files private anyway, but for the a.out
   loader stuff we need to do it earlier */

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;
```

3、这里先为读入的二进制文件注册到 elf 进程文件表中（elf_exec_fileno）。随后初始化 bss 段、代码段和数据段的起始地址和终止地址。随后我们要跳过很长一段代码，这段代码和 elf 链接器（elf interpretor）的加载相关，我们这里不予关注。



4、*flush_old_exec* 完成了当前进程替换成新进程的过程。可以看到其内部调用 exec_mmap 完成当前进程内存的释放，我们可以看下 exec_mmap 具体的实现细节。首先调用 mm_release 释放掉当前进程所占用的内存（old_mm），并且将当前进程的内存空间替换成 bprm→mm 所指定的页面，而这块空间，便是新进程在初始化时暂时向内核借用的存储空间，当这段空间读取到 current→mm 以后，事实上也就完成了旧进程到新进程的替换。这个时候 bprm→mm 这块内核空间也就完成了它的使命，于是被置为 NULL 予以回收。

```
/* 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;

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->rss = 0;
current->mm->free_area_cache = current->mm->mmap_base;
retval = setup_arg_pages(bprm, STACK_TOP, executable_stack);
if (retval < 0) {
    send_sig(SIGKILL, current, 0);
    goto out_free_dentry;
}

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

5、这个片段就是完成对 current→mm 的初始化和实际将参数映射到用户空间的操作，注意由于刚才的 *flust_old_exec* 的逻辑，这里 current 已经指向新的进程。最后一行将新进程用户堆栈的起点定位在 bprm→p，也就是紧随在环境变量和用户变量所占用的内存之后。这里有一个重要的函数，也就是 *setup_arg_pages*，这个函数的作用显而易见，就是将之前暂存到内核空间的参数 bprm→pages 映射到用户空间，同时释放内核空间。我们来看看这个函数做了什么工作。

```
int setup_arg_pages(struct linux_binprm *bprm, int executable_stack)
{
        unsigned long stack_base;
        struct vm_area_struct *mpnt;
        struct mm_struct *mm = current->mm;
        int i;
        long arg_size;
        stack_base = STACK_TOP - MAX_ARG_PAGES * PAGE_SIZE;
    //一般的堆栈向下增长，那么参数最大能增长到的地方就是 stack_base，因为MAX_ARG_PAGES 限制了参数的总页 数，不过这个值已经够大了。
        mm->arg_start = bprm->p + stack_base;
    //将参数的起始地址调整为实际的起始地址，根据就是 bprm 的字段 p，在拷贝参数进内核的时候已经将 p 置为了 PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *)减去参数的总大小，那么这里的 mm->arg_start 正好是参数的从低地址到高地址的起始地址。
        arg_size = STACK_TOP - (PAGE_MASK & (unsigned long) mm->arg_start);
    //参数的大小，由于参数还是以页的形式进行管理的，那么和页掩码（PAGE_MASK:0xfffff000）相与事实上就是将页内偏移屏蔽掉，参数大小是页的整数倍，从而这部分也是无须计算的
        bprm->p += stack_base;
        if (bprm->loader)
                bprm->loader += stack_base;
        bprm->exec += stack_base;
        mpnt = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    //为新的进程分配的第一个 vma，主要就是设置参数，实际上最终就是 main 函数的参数
        if (!mpnt)
                return -ENOMEM;
        if (security_vm_enough_memory(arg_size >> PAGE_SHIFT)) {
                kmem_cache_free(vm_area_cachep, mpnt);
                return -ENOMEM;
        }
        memset(mpnt, 0, sizeof(*mpnt));
        down_write(&mm->mmap_sem);
       {
                mpnt->vm_mm = mm;
                mpnt->vm_start = PAGE_MASK & (unsigned long) bprm->p;
              //由于 bprm→p 加上了 stack_base，这个时候它指向了参数的真正起始地址
              mpnt->vm_end = STACK_TOP;
                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];
            insert_vm_struct(mm, mpnt);
            mm->stack_vm = mm->total_vm = vma_pages(mpnt);
        }
        for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
    //这个循环将 bprm 的 page 映射到了新的地址空间
            struct page *page = bprm->page[i];
            if (page) {
                bprm->page[i] = NULL; //释放内核空间中相应的页
                install_arg_page(mpnt, page, stack_base); //具体映射，就是建立页表映射
            }
            stack_base += PAGE_SIZE;
        }
        up_write(&mm->mmap_sem);
        return 0;
}
```

6、从这里开始，系统已经为 elf 的加载做了最充分的准备，那么下面完成的工作自然是针对程序段信息的载入。当然这段代码也是 *load_elf_binary* 中最重要的部分。我们来看具体的代码片段

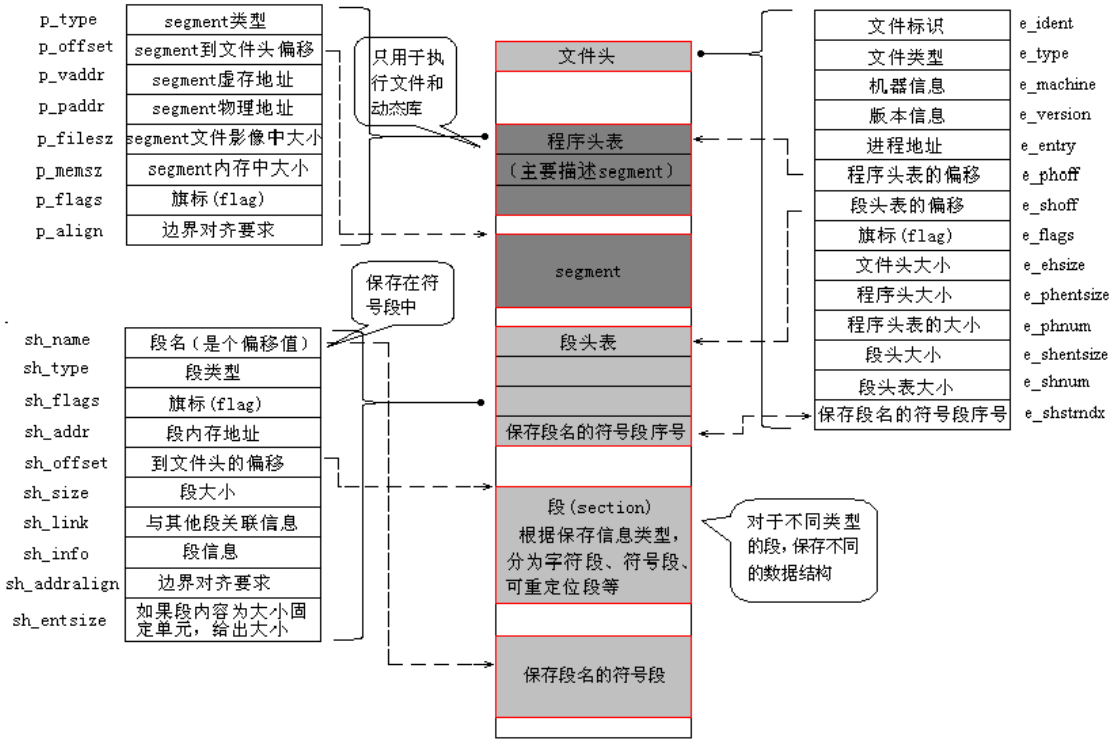
```
        for(i = 0, elf_ppnt = elf_phdata; i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
            int elf_prot = 0, elf_flags;
            unsigned long k, vaddr;
            if (elf_ppnt->p_type != PT_LOAD)
                continue;
            if (unlikely (elf_brk > elf_bss)) {
                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
```

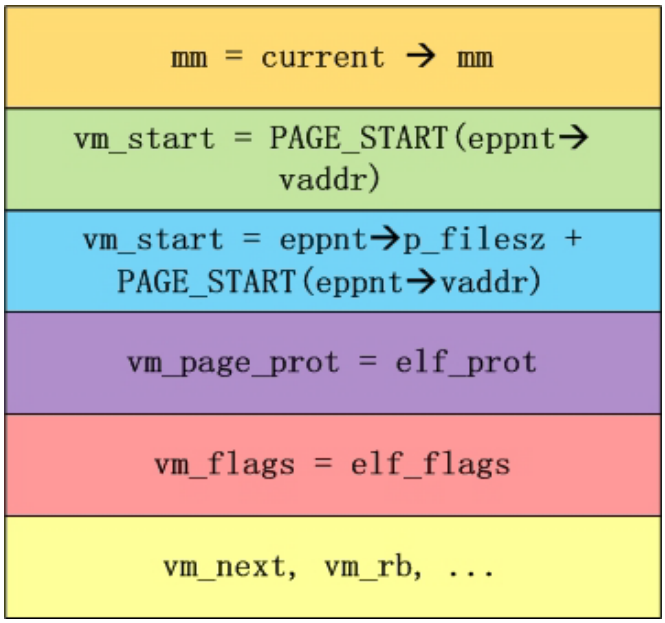
```
                 * <= p_memsz so it is only necessary to check p_memsz.
                 */
                if (k > TASK_SIZE || elf_ppnt->p_filesz > elf_ppnt->p_memsz ||
                        elf_ppnt->p_memsz > TASK_SIZE ||
                        TASK_SIZE - elf_ppnt->p_memsz < k) {
                        /* 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;
        }
```

7、这段代码在目标映像（elf 文件）的程序头表中搜索，寻找类型为 PT_LOAD 的部分（elf 文件中的各种段（segment）信息）进行进一步处理。为了更加清晰地描述上述代码的逻辑，下图先给出 elf 文件主要结构。



这里先假定装入地址是固定的，然后再根据映像是否允许浮动而作出调整。具体 片头数据结构中的 *p_vaddr* 提供了映像在连接时确定的装入地址 *vaddr*。如果映像的类型为 *ET_EXEC*，(或者 load_addr_set 已经被设置成 1，见下)那么装入地址就是固定的。而若类型为 *ET_DYN*（即共享库），那么即使装入地址固定也要加上一个偏移量，代码中给出了计算方法，其中 *ELF_ET_DYN_BASE* 对于 x86 定义为(*TASK_SIZE* / 3 * 2)，而 *ELF_PAGESTART* 表示按页面边界对齐。

确定了装入地址之后，就通过 *elf_map* 建立用户空间虚存区间与目标映像文件中某个连续区间之间的映射。elf_map 内部调用了 *do_mmap* 进行相应的程序段（*PT_LOAD* 类型）到 *vm_area* 的映射工作（参考前面对于 *do_mmap* 的分析）。可以看下传入的地址，完成映射之后，该程序段映射的 vm_area 大致如下：



对于类型为 *ET_EXEC* 的可执行程序映像而言，代码中的 *load_bias* 是 0，所以装入的起点就是映像自己提供的地址 *vaddr*。另一方面，对于 *ET_EXEC*，由于参数中的 *elf_flags* 中的 *MAP_FIXED* 标志位为 1，所以给定的映射地址是刚性的而不容许变通，如果与已经映射的区间有冲突就以失败告终。当前段被载入之后，重新调整 *elf_bss*，*elf_brk*，*end_data* 和 *end_code* 的值(为什么每次都要调整呢？)。

跳出这个 for 循环体，所有的程序段的载入也即完成，随后对 current→mm 的 start_code，end_code 等字段的重新赋值。下面是代码片段，到这里，对于 elf 文件的载入（包括之前对可执行文件运行环境准备工作）的分析基本上可以告一段落了。

```
  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;
  }
```

.....

```
set_binfmt(&elf_format);

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;
```

参考文献

[1] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, 3rd ed. USA: O'Reily, November 2005.
[2] Linux内核源代码情景分析，毛德操，胡希朗，第一版．浙江大学出版社, 2001年9月1日.
[3] J.L.Hennessy and D.A.Patterson, Computer Architecture: A Quantitative Approach, 4th ed. San Francisco, USA: Morgan-Kaufmann, 2007.
[4] ericxiao.cublog.cn, linux进程管理之可执行文件的加载和运行[1][2], http://www.cublog.cn/u1/51562/showart_527411.html;
[5] dog250 @ CSDN blog, linux程序的命令行参数, http://blog.csdn.net/dog250/archive/2010/02/09/5303629.aspx;
[6] dog250 @ CSDN blog, fork,vfork以及exec的意义, http://blog.csdn.net/dog250/archive/2010/02/09/5303625.aspx;
[7] 滕启明, ELF文件格式分析, PKU/SSDB-03-TN-005, 2003年5月;