第35回 | 扒开 execve 的皮

Original 闪客 低并发编程 2022-05-06 17:30 Posted on 北京

收录于合集
#操作系统源码

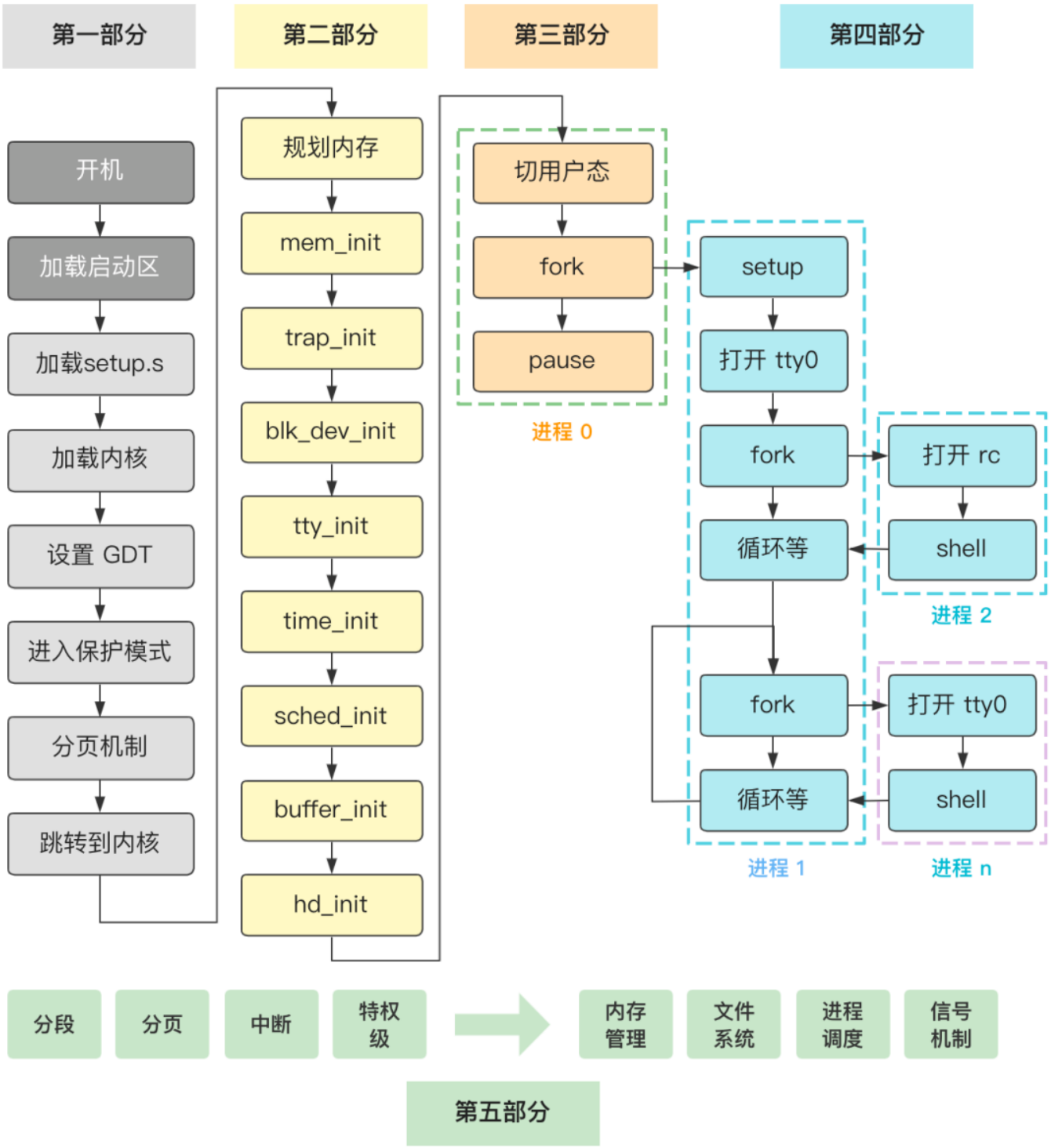
43个

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。

本系列的 GitHub 地址如下，希望给个 star 以示鼓励（文末**阅读原文**可直接跳转，也可以将下面的链接复制到浏览器里打开）
https://github.com/sunym1993/flash-linux0.11-talk

本回的内容属于第四部分。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

第一部分 进入内核前的苦力活

第二部分 大战前期的初始化工作

第三部分：一个新进程的诞生

第四部分：shell 程序的到来

第34回 | 进程2的创建
第35回 | 扒开 execve 的皮（本文）

------- 正文开始 -------

书接上回，上回书咱们说到，进程 1 再次通过 fork 函数创建了进程 2，且进程 2 通过 close 和 open 函数，将 0 号文件描述符指向的标准输入 /dev/tty0 更换为指向 /etc/rc 文件。

```
void init(void) {
    ...
    if (!(pid=fork())) {
        close(0);
        open("/etc/rc",O_RDONLY,0);
        execve("/bin/sh",argv_rc,envp_rc);
        _exit(2);
    }
    ...
}
```

此时进程 2 和进程 1 几乎是完全一样的。

接下来进程 2 就将变得不一样了，会通过一个经典的，也是最难理解的 **execve** 函数调用，使自己摇身一变，成为 **/bin/sh** 程序继续运行！

我们先打开 execve，开一下它的调用链。

```
static char * argv_rc[] = { "/bin/sh", NULL };
static char * envp_rc[] = { "HOME=/", NULL };

// 调用方
execve("/bin/sh",argv_rc,envp_rc);

// 宏定义
_syscall3(int,execve,const char *,file,char **,argv,char **,envp)

// 通过系统调用进入到这里
EIP = 0x1C
_sys_execve:
    lea EIP(%esp),%eax
    pushl %eax
    call _do_execve
    addl $4,%esp
    ret

// 最终执行的函数
int do_execve(
        unsigned long * eip,
        long tmp,
        char * filename,
        char ** argv,
        char ** envp) {
    ...
}
```

我们在 第25回 | 通过 fork 看一次系统调用 已经详细分析了整个调用链中的栈以及参数传递的过程。

所以这里我们就不再赘述，直接把这里的参数传过来的样子写出来。

eip 调用方触发系统调用时由 CPU 压入栈空间中的 eip 的指针 。
tmp 是一个无用的占位参数。
filename 是 "/bin/sh"
argv 是 { "/bin/sh", NULL }
envp 是 { "HOME=/", NULL }

好了，接下来我们看看整个 **do_execve** 函数，它非常非常长！我先把整个结构列出。

```
int do_execve(...) {
    // 检查文件类型和权限等
    ...
    // 读取文件的第一块数据到缓冲区
    ...
    // 如果是脚本文件，走这里
    if (脚本文件判断逻辑) {
        ...
    }
    // 如果是可执行文件，走这里
    // 一堆校验可执行文件是否能执行的判断
    ...
    // 进程管理结构的调整
    ...
    // 释放进程占有的页面
    ...
    // 调整线性地址空间、参数列表、堆栈地址等
    ...
    // 设置 eip 和 esp，这里是 execve 变身大法的关键！
    eip[0] = ex.a_entry;
    eip[3] = p;
    return 0;
    ...
}
```

整理起来的步骤就是。

1. 检查文件类型和权限等
2. 读取文件的第一块数据到缓冲区
3. 脚本文件与可执行文件的判断
4. 校验可执行文件是否能执行
5. 进程管理结构的调整
6. 释放进程占有的页面
7. 调整线性地址空间、参数列表、堆栈地址等
8. 设置 eip 和 esp，完成摇身一变

如果去掉一些逻辑校验和判断，那核心逻辑就是**加载文件**、**调整内存**、**开始执行**三个步骤，由于这些部分的内容已经非常复杂了，所以我们就去掉那些逻辑校验的部分，直接挑主干逻辑进行讲解，以便带大家认清 execve 的本质。

走你~

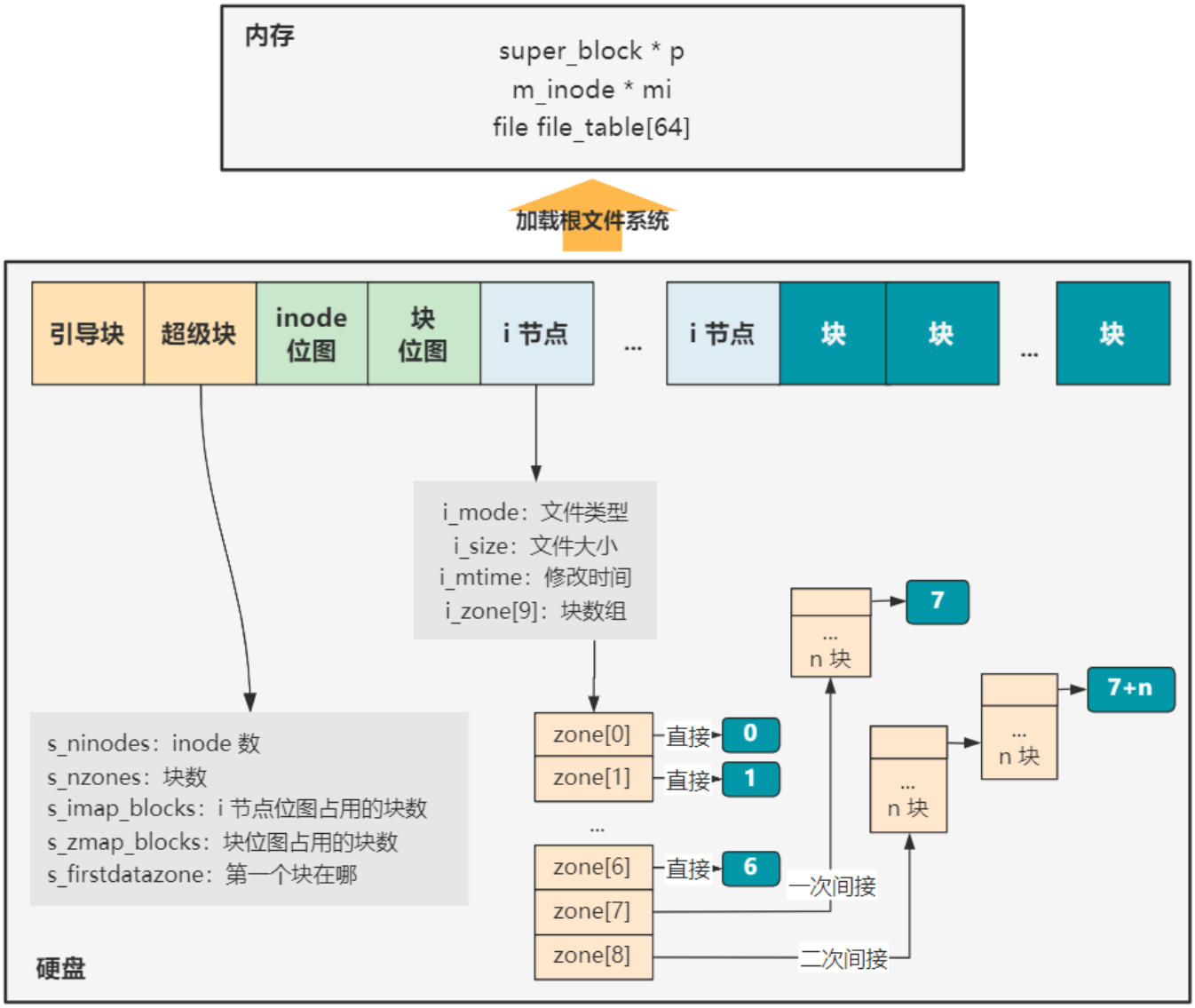
读取文件开头 1KB 的数据

先是根据文件名，找到并读取文件里的内容

```
// exec.c
int do_execve(...) {
    ...
    // 根据文件名 /bin/sh 获取 inode
    struct m_inode * inode = namei(filename);
    // 根据 inode 读取文件第一块数据（1024KB）
    struct buffer_head * bh = bread(inode->i_dev,inode->i_zone[0]);
    ...
}
```

很简单，**就是读取了文件（/bin/sh）第一个块**，也就是 1KB 的数据，

在 第32回 | 加载根文件系统 里说过文件系统的结构，所以代码里 **inode -> i_zone[0]** 就刚好是文件开头的 1KB 数据。



OK，现在这 1KB 的数据，就已经在内存中了，但还没有解析。

解析这 1KB 的数据为 exec 结构

接下来的工作就是解析它，本质上就是按照指定的数据结构来解读罢了。

```
// exec.c
int do_execve(...) {
    ...
    struct exec ex = *((struct exec *) bh->b_data);
    ...
}
```

先从刚刚读取文件返回的缓冲头指针中取出数据部分 **bh -> data**，也就是文件前 1024 个字节，此时还是一段读不懂的二进制数据。

然后按照 exec 这个结构体对其进行解析，它便有了生命。

```
struct exec {
    // 魔数
    unsigned long a_magic;
    // 代码区长度
    unsigned a_text;
    // 数据区长度
    unsigned a_data;
    // 未初始化数据区长度
    unsigned a_bss;
    // 符号表长度
    unsigned a_syms;
    // 执行开始地址
    unsigned a_entry;
    // 代码重定位信息长度
    unsigned a_trsize;
    // 数据重定位信息长度
    unsigned a_drsize;
};
```

上面的代码就是 exec 结构体，这是 **a.out** 格式文件的头部结构，现在的 Linux 已经弃用了这种古老的格式，改用 ELF 格式了，但大体的思想是一致的。

这个结构体里的字段表示什么，等我们用到了再说，你可以先通过我的注释自己体会下。

判断是脚本文件还是可执行文件

我们写一个 Linux 脚本文件的时候，通常可以看到前面有这么一坨东西。

```
#!/bin/sh
#!/usr/bin/python
```

你有没有想过为什么我们通常可以直接执行这样的文件？其实逻辑就在下面这个代码里。

```
// exec.c
int do_execve(...) {
    ...
    if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') {
        ...
    }
    brelse(bh);
    ...
}
```

可以看到，很简单粗暴地判断前面两个字符是不是 **#!**，如果是的话，就走**脚本文件**的执行逻辑。

当然，我们现在的 /bin/sh 是个**可执行的二进制文件**，不符合这样的条件，所以这个 if 语句里面的内容我们也可以不看了，直接看外面，执行可执行二进制文件的逻辑。

第一步就是 brelse 释放这个缓冲块，因为已经把这个缓冲块内容解析成 exec 结构保存到我们程序的栈空间里了，那么这个缓冲块就可以释放，用于其他读取磁盘时的缓冲区。

不重要，我们继续往下看。

准备参数空间

我们执行 /bin/sh 时，还给它传了 argc 和 envp 参数，就是通过下面这一系列代码来实现的。

```
#define PAGE_SIZE 4096
#define MAX_ARG_PAGES 32

// exec.c
int do_execve(...) {
    ...
    // p = 0x1FFFC = 128K - 4
    unsigned long p = PAGE_SIZE * MAX_ARG_PAGES - 4;
    ...
    // p = 0x1FFF5 = 128K - 4 - 7
    p = copy_strings(envc,envp,page,p,0);
    // p = 0x1FFED = 128K - 4 - 7 - 8
    p = copy_strings(argc,argv,page,p,0);
    ...
    // p = 0x3FFFFED = 64M - 4 - 7 - 8
    p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
    // p = 0x3FFFFD0
    p = (unsigned long) create_tables((char *)p,argc,envc);
    ...
    // 设置栈指针
    eip[3] = p;
}
```

准备参数空间的过程，同时也伴随着一个表示地址的 unsigned long p 的计算轨迹。

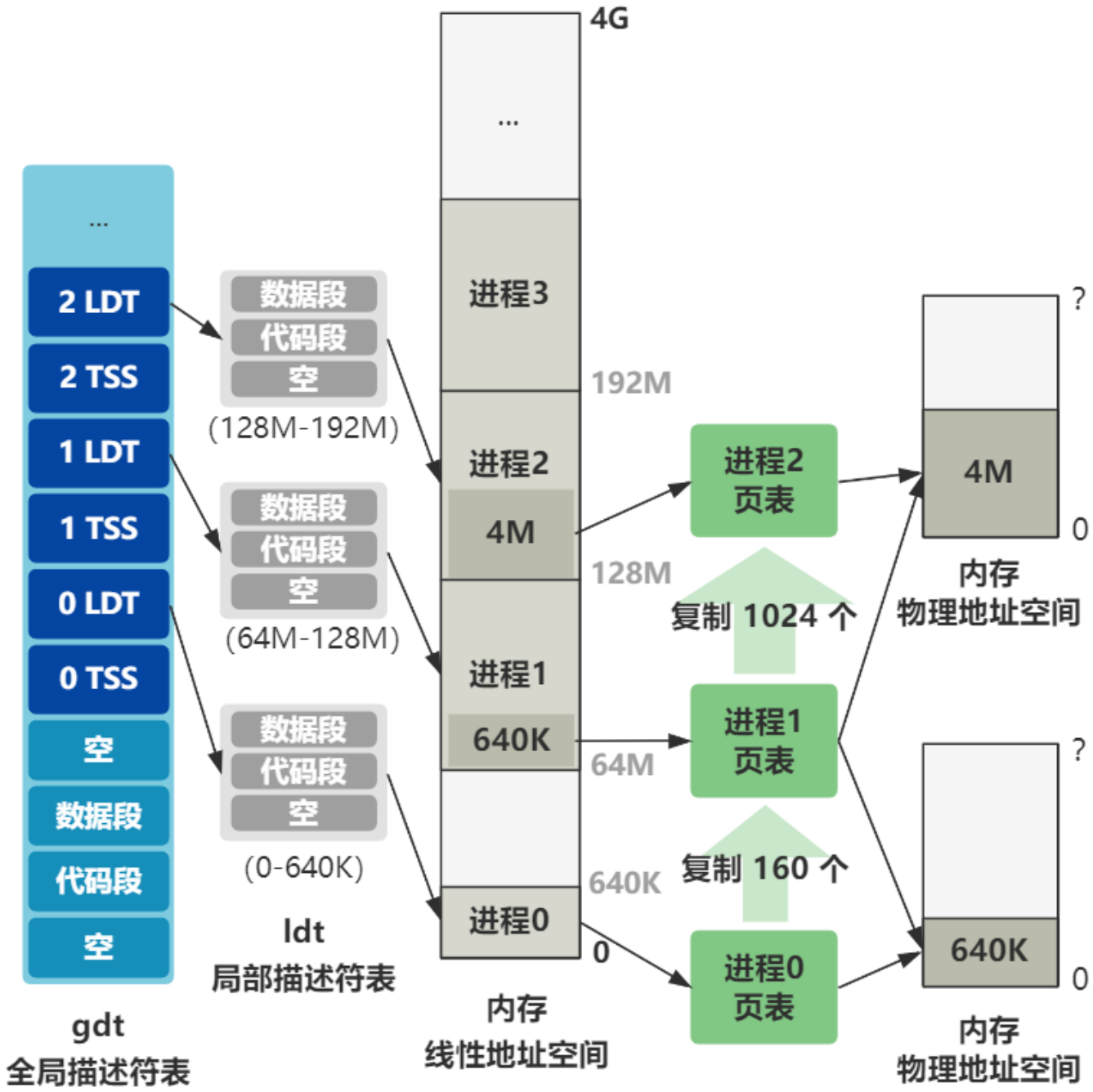
有点难以理解，别急，我们一点点分析就会恍然大悟。

开头一行计算出的 p 值为

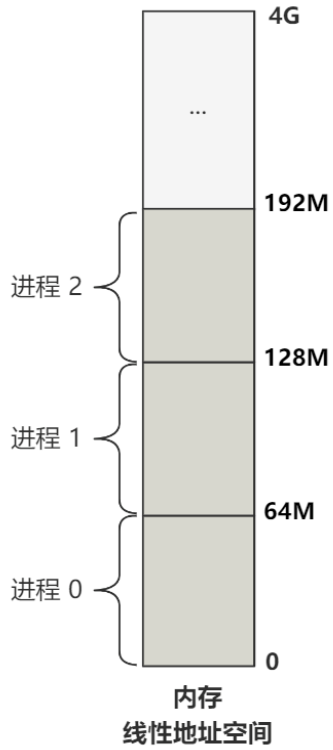
p = 4096 * 32 - 4 = 0x20000 - 4 = 128K - 4

为什么是这个数呢？整个这块讲完你就会知道，这表示**参数表**，每个进程的参数表大小为**128K**，在每个进程地址空间的**最末端**。

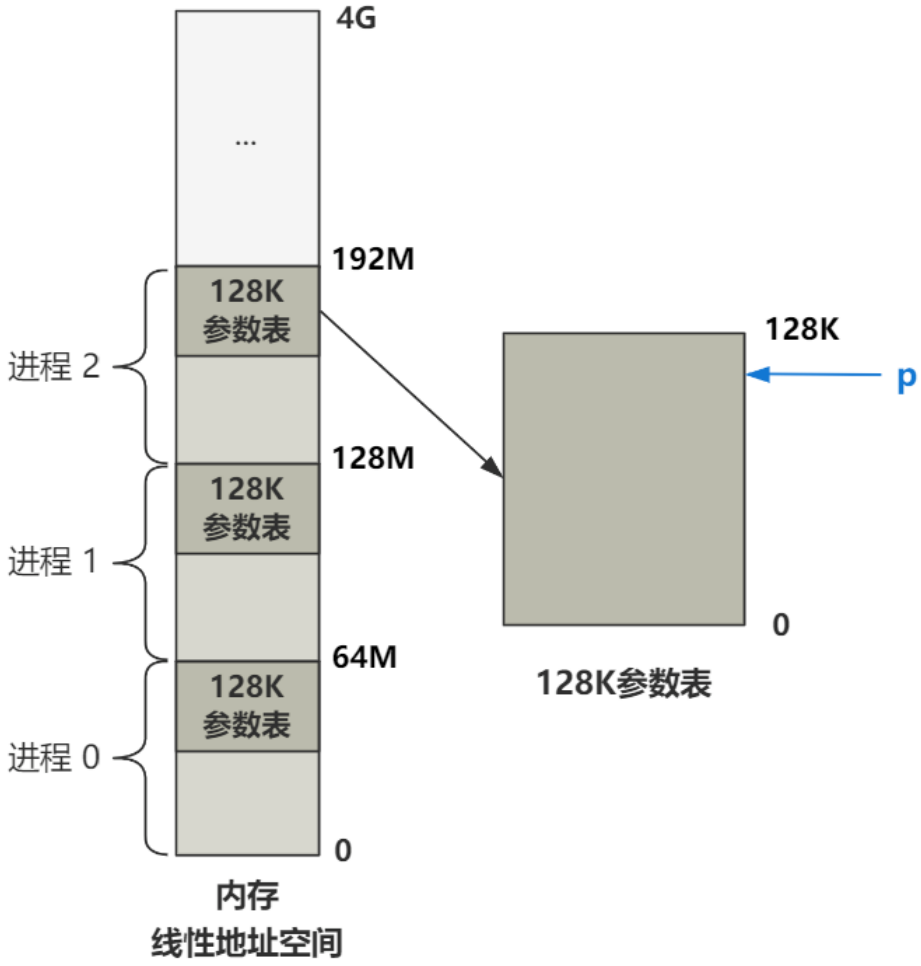
还记得之前的一张图么？



我们说过，**每个进程通过不同的局部描述符在线性地址空间中瓜分出不同的空间**，一个进程占**64M**，我们单独把这部分表达出来。



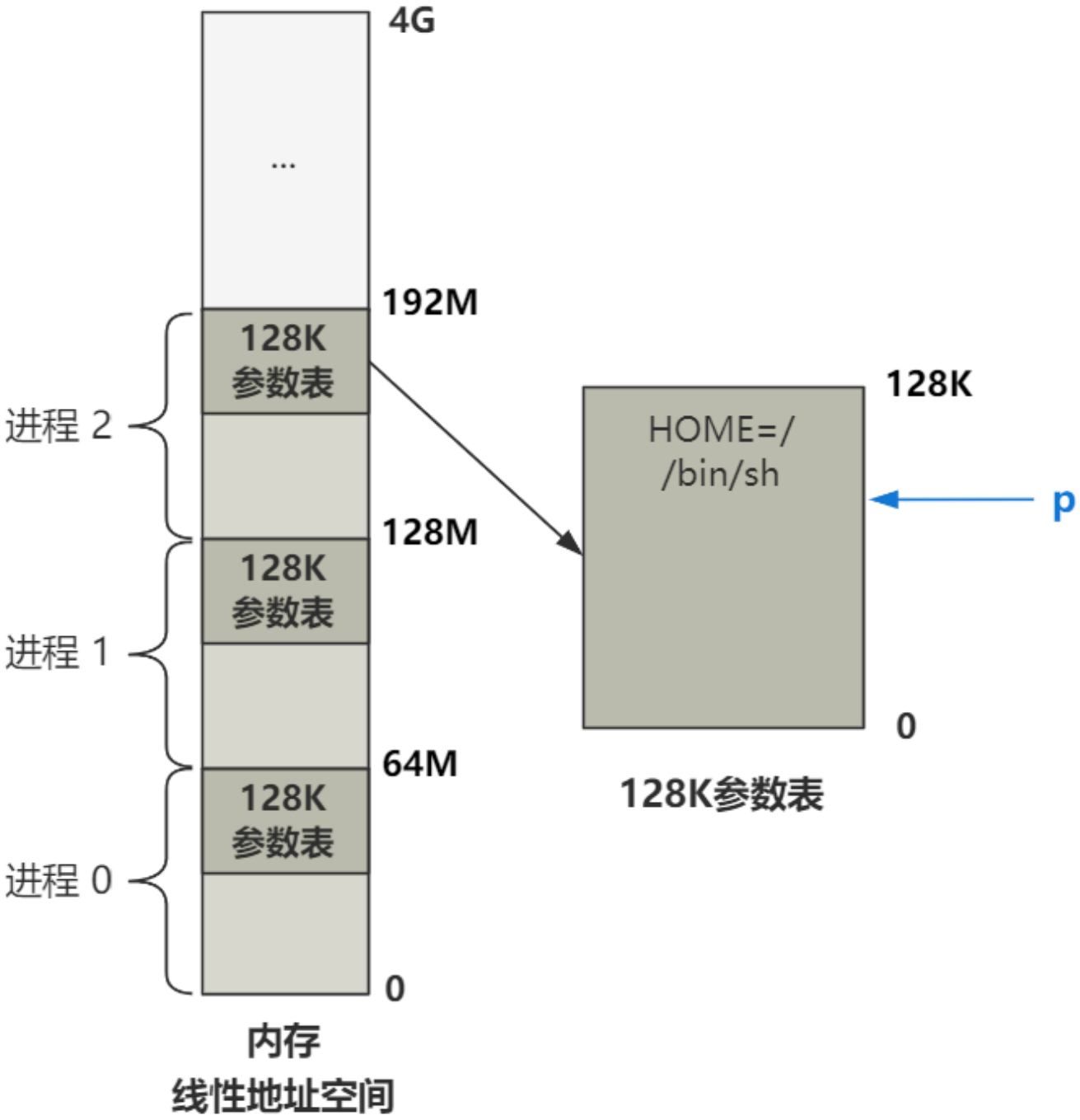
参数表为 128K，就表示每个进程的线性地址空间的末端 128K，是为参数表保留的，目前这个 p 就指向了参数表的开始处（偏移 4 字节）。



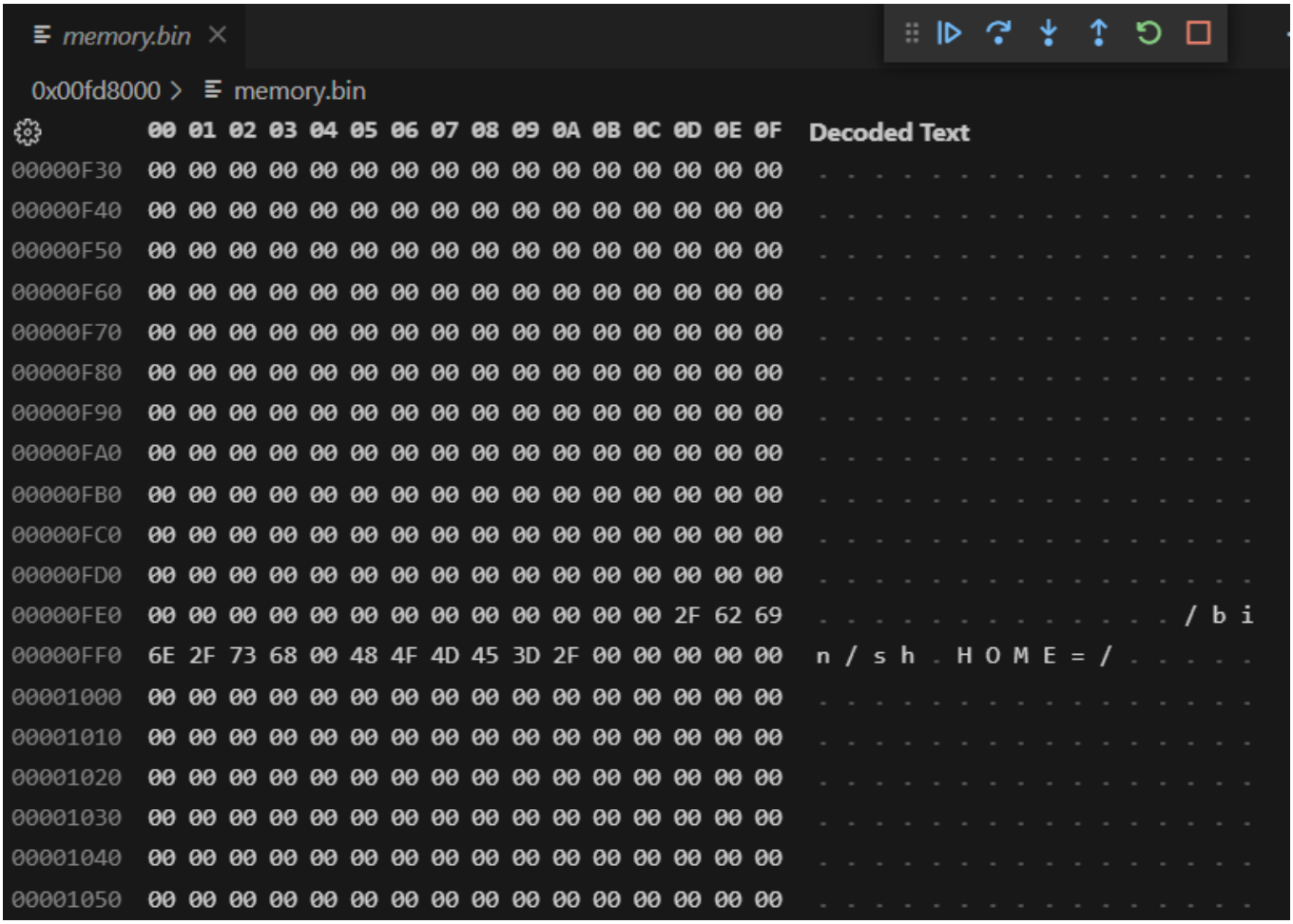
接下来两个 **copy_strings** 就是往这个参数表里面存放信息，不过具体存放的只是字符串常量值的信息，随后他们将被引用，有点像 Java 里 class 文件的字符串常量池思想。

```
// exec.c
int do_execve(...) {
    ...
    // p = 0x1FFF5 = 128K - 4 - 7
    p = copy_strings(envc,envp,page,p,0);
    // p = 0x1FFED = 128K - 4 - 7 - 8
    p = copy_strings(argc,argv,page,p,0);
    ...
}
```

具体说来，**envp** 表示字符串参数 **"HOME=/"**，**argv** 表示字符串参数 **"/bin/sh"**，两个 copy 就表示把这个字符串参数往参数表里存，相应地指针 p 也往下移动（共移动了 7 + 8 = 15 个字节），和压栈的效果是一样的。



当然，这个只是示意图，实际上这些字符串都是紧挨着的，我们通过 debug 查看参数表位置处的内存便可以看到真正存放的方式。



可以看到，两个字符串乖乖地被安排在了参数表内存处，且参数与参数之间用 00 也就是 NULL 来分隔。

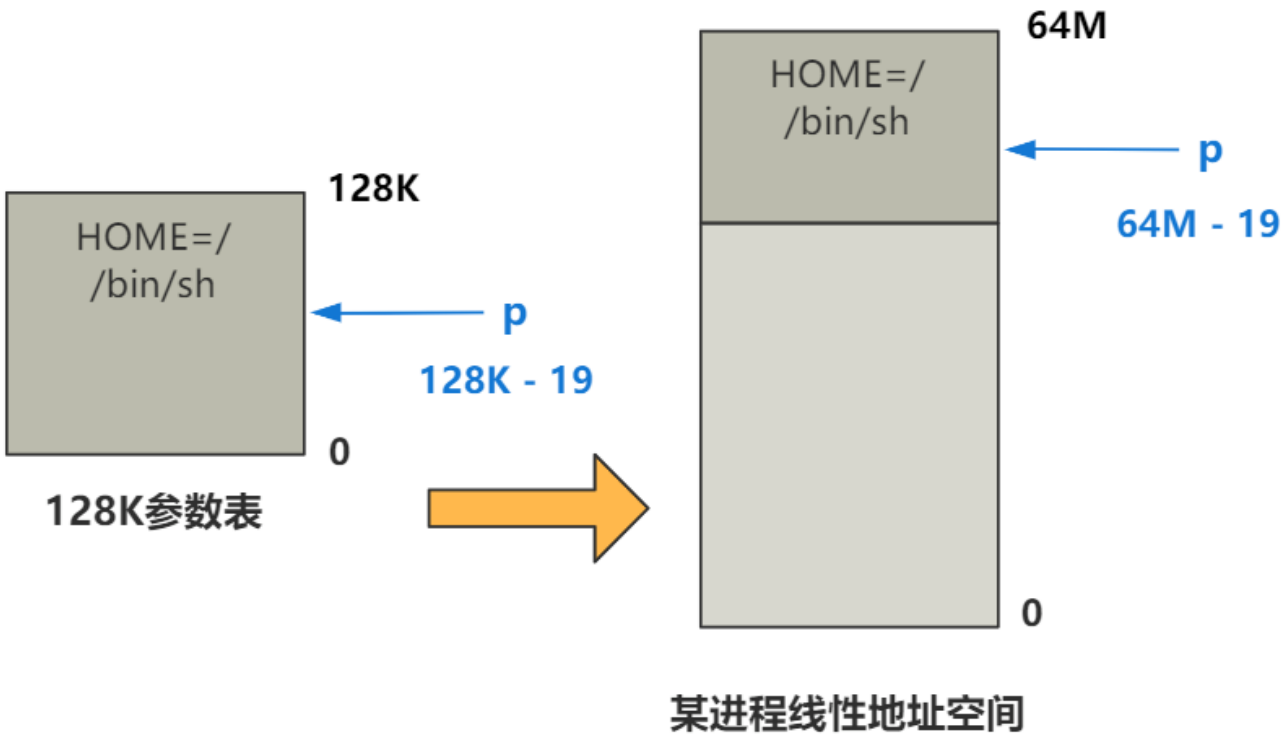
接下来是**更新局部描述符。**

```
#define PAGE_SIZE 4096
#define MAX_ARG_PAGES 32

// exec.c
int do_execve(...) {
    ...
    // p = 0x3FFFFED = 64M - 4 - 7 - 8
    p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
    ...
}
```

很简单，就是根据 **ex.a_text** 修改局部描述符中的**代码段限长** code_limit，其他没动。

ex 结构里的 a_text 是生成 /bin/sh 这个 a.out 格式的文件时，写在头部的值，用来表示代码段的长度。至于具体是怎么生成的，我们无需关心。

由于这个函数返回值是数据段限长，也就是 64M，所以最终的 p 值被调整为了以每个进程的线性地址空间视角下的地址偏移，大家可以仔细想想怎么算的。



接下来就是真正**构造参数表**的环节了。

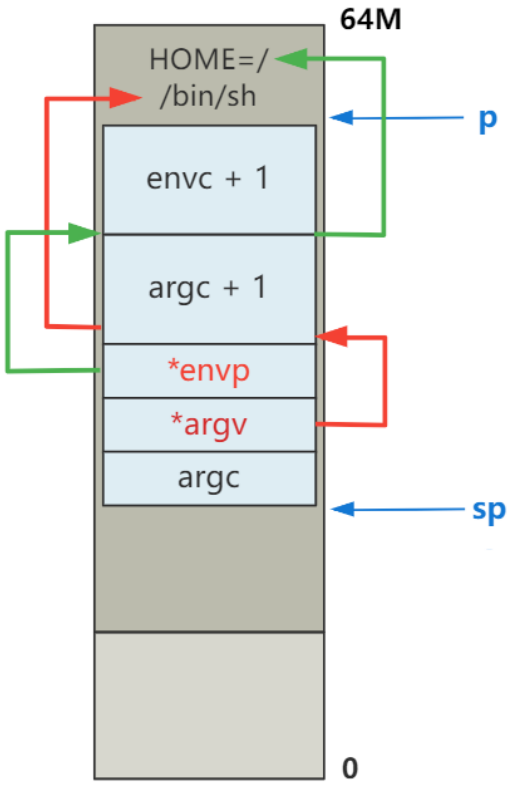
```
#define PAGE_SIZE 4096
#define MAX_ARG_PAGES 32

// exec.c
int do_execve(...) {
    ...
    // p = 0x3FFFFD0
    p = (unsigned long) create_tables((char *)p,argc,envc);
    ...
}
```

刚刚仅仅是往参数表里面丢入了需要的字符串常量值信息，现在就需要真正把参数表构建起来。

我们展开 **create_tables**。

```
/*
 * create_tables() parses the env- and arg-strings in new user
 * memory and creates the pointer tables from them, and puts their
 * addresses on the "stack", returning the new stack pointer value.
 */
static unsigned long * create_tables(char * p,int argc,int envc) {
    unsigned long *argv,*envp;
    unsigned long * sp;

    sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
    sp -= envc+1;
    envp = sp;
    sp -= argc+1;
    argv = sp;
    put_fs_long((unsigned long)envp,--sp);
    put_fs_long((unsigned long)argv,--sp);
    put_fs_long((unsigned long)argc,--sp);
    while (argc-->0) {
        put_fs_long((unsigned long) p,argv++);
        while (get_fs_byte(p++)) /* nothing */ ;
    }
    put_fs_long(0,argv);
    while (envc-->0) {
        put_fs_long((unsigned long) p,envp++);
        while (get_fs_byte(p++)) /* nothing */ ;
    }
    put_fs_long(0,envp);
    return sp;
}
```

可能稍稍有点烧脑，不过如果你一行一行仔细分析，不难分析出就是把参数表空间变成了如下样子。



某进程线性地址空间

最后，将 sp 返回给 p，这个 p 将作为一个新的栈顶指针，给即将要完成替换的 /bin/sh 程序，也就是下面的代码。

```
// exec.c
int do_execve(...) {
    ...
    // 设置栈指针
    eip[3] = p;
}
```

为什么这样操作就可以达到更换栈顶指针的作用呢？那我们结合着更换代码指针 PC 来进行讲解。

设置 eip 和 esp，完成摇身一变

下面这两行就是 execve 完成摇身一变的关键，解释了它为什么能做到变成一个新程序开始执行的关键密码。

```
// exec.c
int do_execve(unsigned long * eip, ...) {
    ...
    eip[0] = ex.a_entry;
    eip[3] = p;
    ...
}
```

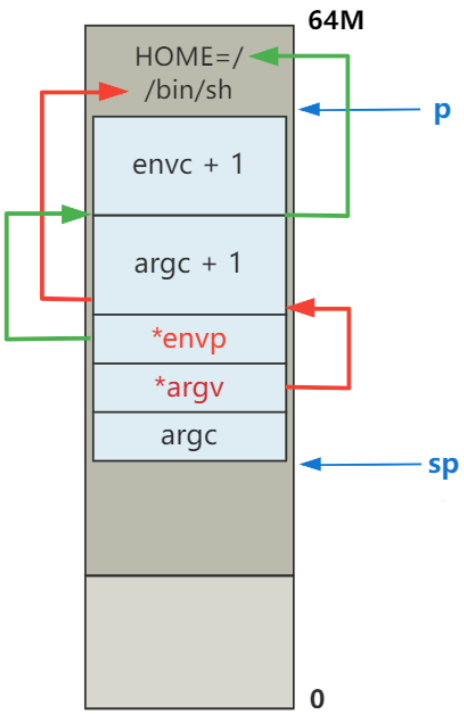
什么叫一个新程序开始执行呢?

其实本质上就是，**代码指针 eip 和栈指针 esp 指向了一个新的地方。**

代码指针 eip 决定了 CPU 将执行哪一段指令，栈指针 esp 决定了 CPU 压栈操作的位置，以及读取栈空间数据的位置，在高级语言视角下就是**局部变量**以及**函数调用链的栈帧。**

所以这两行代码，第一行重新设置了**代码指针 eip** 的值，指向 /bin/sh 这个 a.out 格式文件的头结构 exec 中的 a_entry 字段，表示该**程序的入口地址。**

第二行重新设置了**栈指针 esp** 的值，指向了我们经过一路计算得到的 p，也就是图中 sp 的值。将这个值作为新的栈顶十分合理。



某进程线性地址空间

eip 和 esp 都设置好了，那么程序摇身一变的工作，自然就结束了，非常简单。

至于为什么往 eip 的 0 和 3 索引位置处写入数据，就可以达到替换 eip 和 esp 的目的，那我们就得看看这个 eip 变量是怎么来的了。

计算机的世界没有魔法

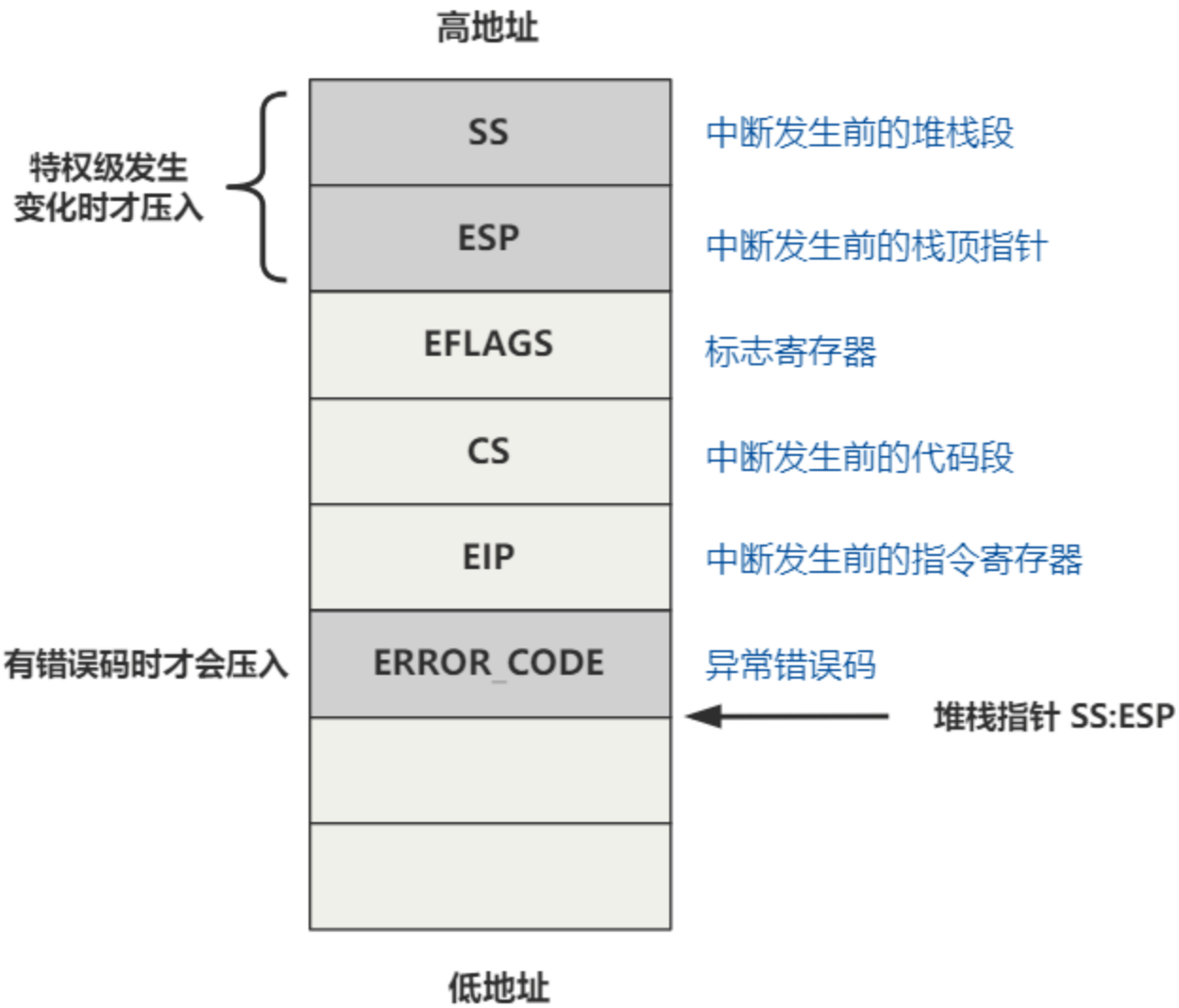
还记得 execve 的调用链么？

```
static char * argv_rc[] = { "/bin/sh", NULL };
static char * envp_rc[] = { "HOME=/", NULL };

// 调用方
execve("/bin/sh",argv_rc,envp_rc);

// 宏定义
_syscall3(int,execve,const char *,file,char **,argv,char **,envp)

// 通过系统调用进入到这里
EIP = 0x1C
_sys_execve:
    lea EIP(%esp),%eax
    pushl %eax
    call _do_execve
    addl $4,%esp
    ret

// exec.c
int do_execve(unsigned long * eip, ...) {
    ...
    eip[0] = ex.a_entry;
    eip[3] = p;
    ...
}
```

千万别忘了，我们这个 **do_execve** 函数，是通过一开始的 **execve** 函数触发了**系统调用**来到

的这里。

系统调用是一种**中断**，前面说过，中断时 CPU 会给栈空间里压入一定的信息，这部分信息是死的，查手册可以查得到。



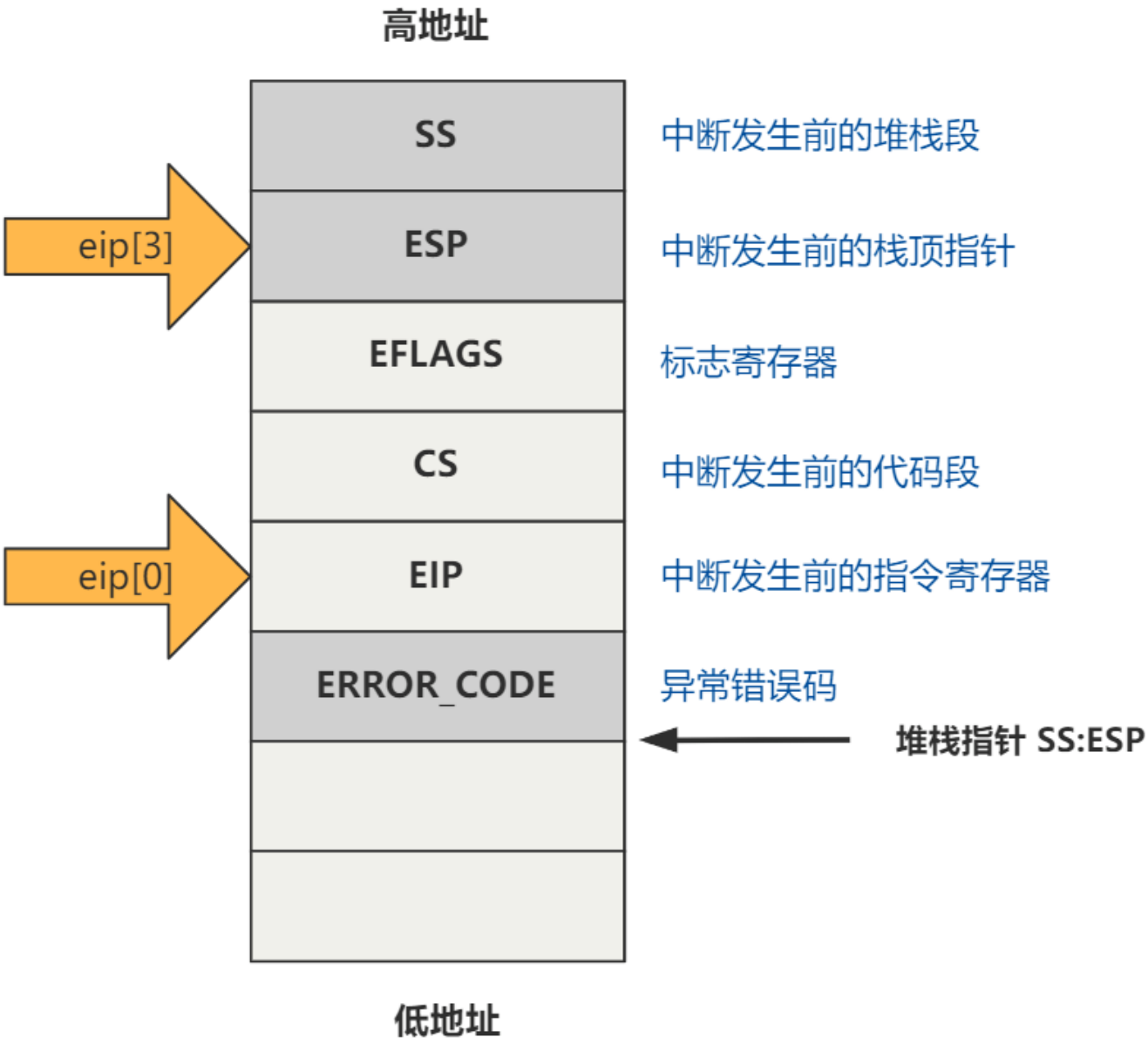
然后，进入中断以后，通过系统调用查表进入到 **_sys_execve** 这里。

```
EIP = 0x1C
_sys_execve:
    lea EIP(%esp),%eax
    pushl %eax
    call _do_execve
    addl $4,%esp
    ret
```

看到没？在真正调用 **do_execve** 函数时，**_sys_execve** 这段代码偷偷地插入了一个小步骤，就是把当前栈顶指针 esp 偏移到 EIP 处的地址值给当做第一个参数 **unsigned long * eip** 传入进来了。

而偏移 EIP 处的位置，恰好就是中断时压入的 EIP 的值的位置，表示中断发生前的指令寄存器的值。

所以 eip[0] 就表示栈空间里的 EIP 位置，eip[3] 就表示栈空间里的 ESP 位置。



由于我们现在处于中断，所以**中断返回**后，也就是 do_execve 这个函数 return 之后，就会寻找中断返回前的这几个值（包括 eip 和 esp）进行恢复。

这里有疑惑的同学，看下我之前写的 认认真真的聊聊中断 和 认认真真的聊聊"软"中断 这两篇文章，我认为把中断的原理彻底讲清楚了，不过其实就是读 CPU 手册罢了。

所以如果我们把这个栈空间里的 eip 和 esp 进行替换，换成执行 /bin/sh 所需要的 eip 和 esp，那么中断返回的**"恢复"**工作，就犹如**"跳转"**到一个新程序那里一样，其实是我们欺骗了 CPU，达到了 execve 这个函数的魔法效果。

所以，**计算机的世界里根本没有魔法**，就是通过这一点点细节而完成的，只是大部分人都不愿意花时间去细究这些细节罢了。

累了吧，休息会

本章讲解了我认为 Linux 0.11 里面最难理解的 execve 函数的核心逻辑，但其实整个顺下来你会发现，它所完成的事情也是由一个个非常简单的事情拼凑起来的。

无非就是把参数表在一个空间里折腾来折腾去给构造好，然后把代码应该从哪里执行的 eip 和新的栈空间应该设置在哪里的 esp 给弄好，就完成了使命。

至于 /bin/sh 文件是怎么构造出来的，那就是 gcc 编译和链接那些事了，而且 Linux 0.11 所用的可执行文件格式 a.out，已经被现在的 ELF 格式所取代，那就更不在我们要研究的范畴，本系列还是要划分好边界问题的，不然就无休止了。

OK，至此，execve 函数就彻底结束了。

```
void init(void) {
    ...
    if (!(pid=fork())) {
        close(0);
        open("/etc/rc",O_RDONLY,0);
        execve("/bin/sh",argv_rc,envp_rc);
        _exit(2);
    }
    ...
}
```

他的返回意味着接下来将执行 /bin/sh 这个可执行文件里的代码。

不过有两个问题：

第一，**/bin/sh** 是躺在磁盘里的可执行文件，并不是 Linux 内核里的代码，所以其实那里面是什么，我们在 Linux 0.11 源码里是找不到的。也就是说，如果磁盘里并没有 /bin/sh 这个文件，Linux 0.11 是启动不起来的，在 open 的时候就会报错宕机了。

第二，我们只将 /bin/sh 文件的头部加载到了内存，其他部分并没有任何代码完成加载这个操作，那接下来跳转到一个并没有加载 /bin/sh 代码的内存时，会发生什么呢？

带着这两个疑问，期待后续的章节吧！

欲知后事如何，且听下回分解。

------- 关于本系列 -------

本系列的开篇词看这，开篇词

本系列的番外故事看这，让我们一起来写本书？也可以直接无脑加入星球，共同参与这场旅行。

最后，本系列**完全免费**，希望大家能多多传播给同样喜欢的人，同时给我的 GitHub 项目点个 star，就在**阅读原文**处，这些就足够让我坚持写下去了！我们下回见。

低并发编程
战略上藐视技术，战术上重视技术
175篇原创内容

Official Account

收录于合集 #操作系统源码 43

上一篇
第34回 | 进程2的创建

下一篇
调试 Linux 最早期的代码

Read more

People who liked this content also liked

今天我下了个JDK
低并发编程

一个合格的Monorepo 应当具备哪些能力?

魔术师卡颂

从Go log库到Zap，怎么打造出好用又实用的Logger

Golang技术分享