第36回 | 缺页中断

Original 闪客 低并发编程 2022-05-11 17:30 Posted on 北京

收录于合集
#操作系统源码

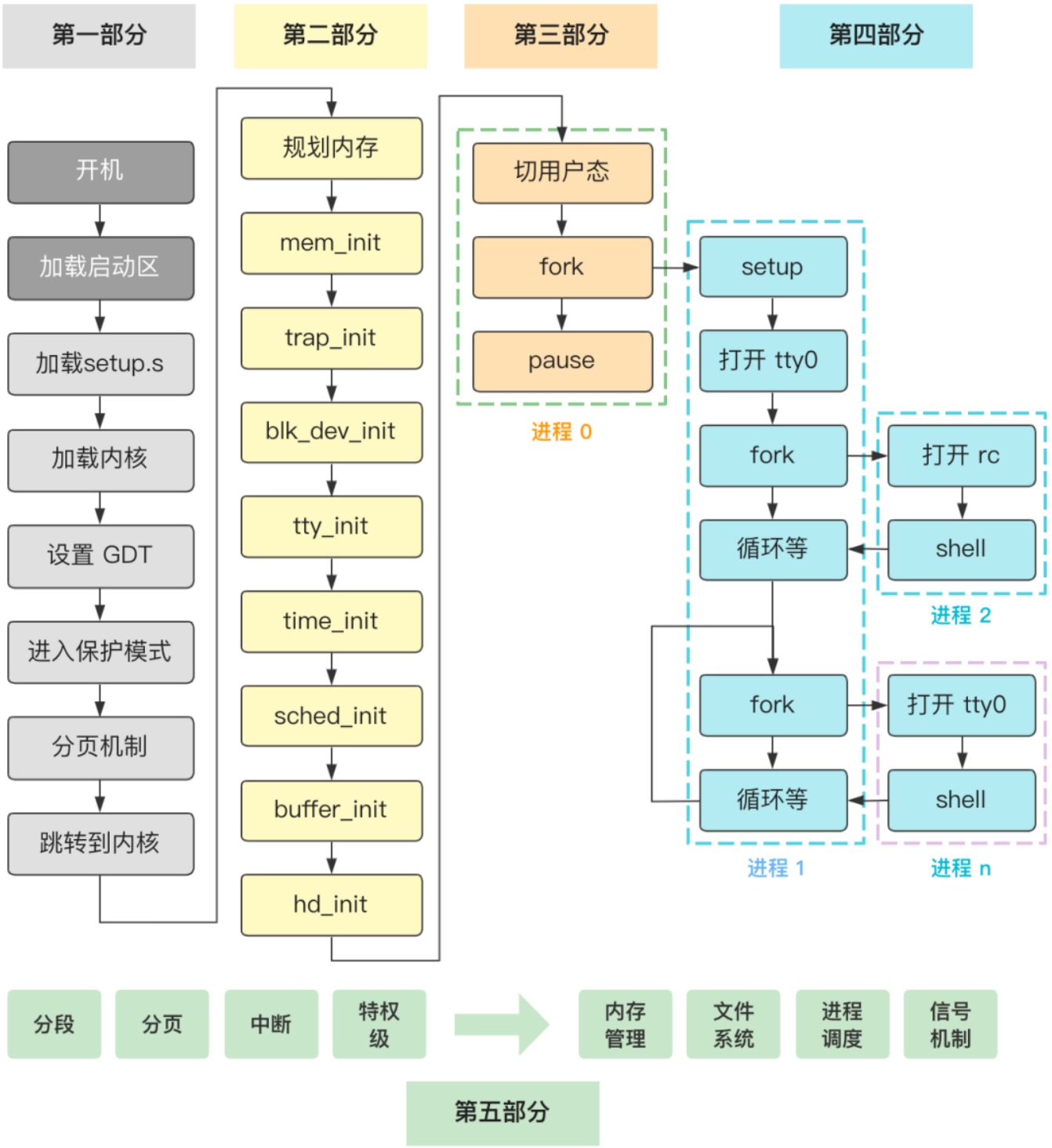
43个

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

------- 正文开始 -------

书接上回，上回书咱们说到，进程 2 通过 **execve** 函数，将自己摇身一变成为 **/bin/sh** 程序，也就是 **shell** 程序开始执行。

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

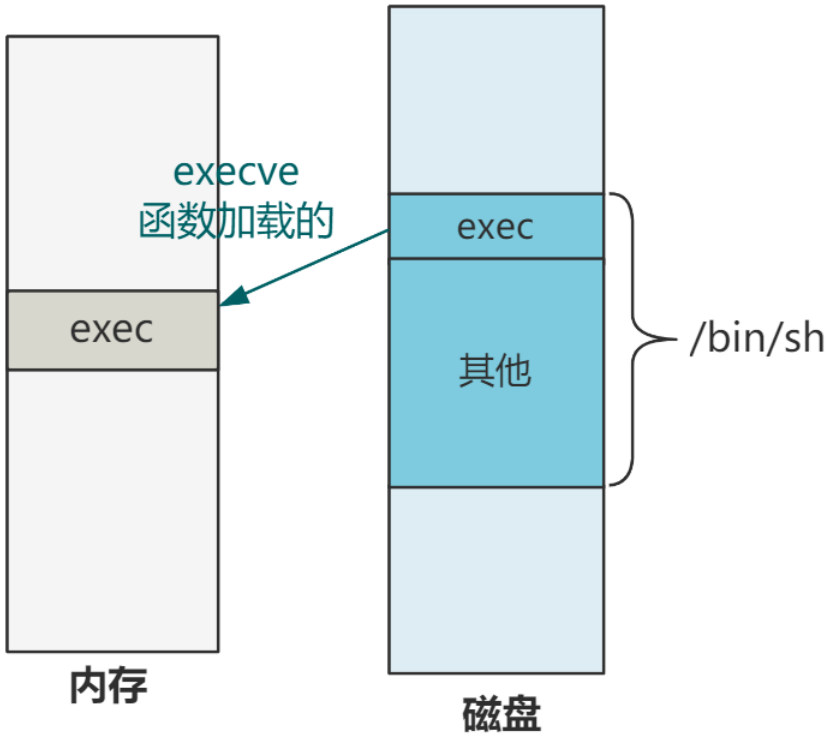
那么此时进程 2 就是 shell 程序了。

再进一步讲，相当于之前的进程 1 通过 **fork + execve** 这两个函数的组合，创建了一个新的进程去加载并执行了 shell 程序。

我们在 Linux 里执行一个程序，比如在命令行中 ./xxx，其内部实现逻辑都是 fork + execve 这个原理。

当然，此时我们仅仅是通过 execve，使得下一条 CPU 指令将会执行到 /bin/sh 程序所在的内存起始位置处，也就是 /bin/sh 头部结构中 **a_entry** 所描述的地址。

但有个问题是，我们仅仅将 /bin/sh 文件的头部加载到了内存，其他部分并没有进行加载，那我们是怎么执行到的 /bin/sh 的程序指令呢？



我们就带着这个问题，开始今天的探索。

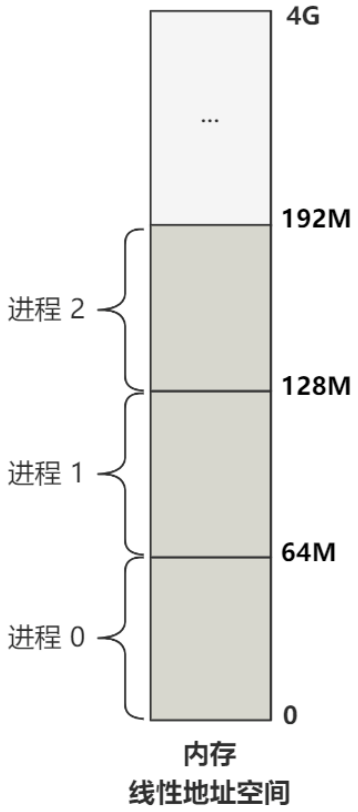
跳转到一个不存在的地址会发生什么

/bin/sh 这个文件并不是 Linux 0.11 源码里的内容，Linux 0.11 只管按照 a.out 这种格式去解读它，跳转到 a.out 格式头部数据结构 **exec.a_entry** 所指向的内存地址去执行指令。

所以这个 a_entry 的值是多少，就完全取决于硬盘中 /bin/sh 这个文件是怎么构造的了，我们简单点，就假设它为 **0**，这表示随后的 CPU 将跳转到 0 地址处进行执行。

当然，这个 0 仅仅表示**逻辑地址**，既没有进行分段，也没有进行分页。

之前说过无数次了，Linux 0.11 的每个进程是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。



由于我们现在所处的代码是属于进程 2，所以逻辑地址 0 通过分段机制映射到线性地址空间，就是 **0x8000000**，表示 **128M** 位置处。

好，128M 这个线性地址，随后将会通过**分页机制**的映射转化为物理地址，这才定位到最终的真实物理内存。

可是，128M 这个线性地址并没有页表映射它，也就是因为上面我们说的，我们除了 /bin/sh 文件的头部加载到了内存外，其他部分并没有进行加载操作。

再准确点说，是 0x8000000 这个线性地址的访问，遇到了页表项的存在位 P 等于 0 的情况。

一旦遇到了这种情况，CPU 会触发一个中断：**页错误（Page-Fault）**，这在 Intel 手册 Volume-3 Chapter 4.7 章节里给出了这个信息。

4.7 PAGE-FAULT EXCEPTIONS

Accesses using linear addresses may cause **page-fault exceptions** (#PF; exception 14). An access to a linear address may cause a page-fault exception for either of two reasons: (1) there is no translation for the linear address; or (2) there is a translation for the linear address, but its access rights do not permit the access.

As noted in Section 4.3, Section 4.4.2, and Section 4.5, there is no translation for a linear address if the translation process for that address would use a paging-structure entry in which the P flag (bit 0) is 0 or one that sets a reserved bit. If there is a translation for a linear address, its access rights are determined as specified in Section 4.6.

When Intel® Software Guard Extensions (Intel® SGX) are enabled, the processor may deliver exception 14 for reasons unrelated to paging. See Section 33.3, "Access-control Requirements" and Section 33.20, "Enclave Page Cache Map (EPCM)" in Chapter 33, "Enclave Access Control and Data Structures." Such an exception is called an **SGX-induced page fault**. The processor uses the error code to distinguish SGX-induced page faults from ordinary page faults.

当然，Page-Fault 在很多情况都会触发，具体是因为什么情况触发的，CPU 会帮我们保存在中断的出错码 **Error Code** 里，这在随后的 Figure 4-12 中给出了详细的出错码说明。

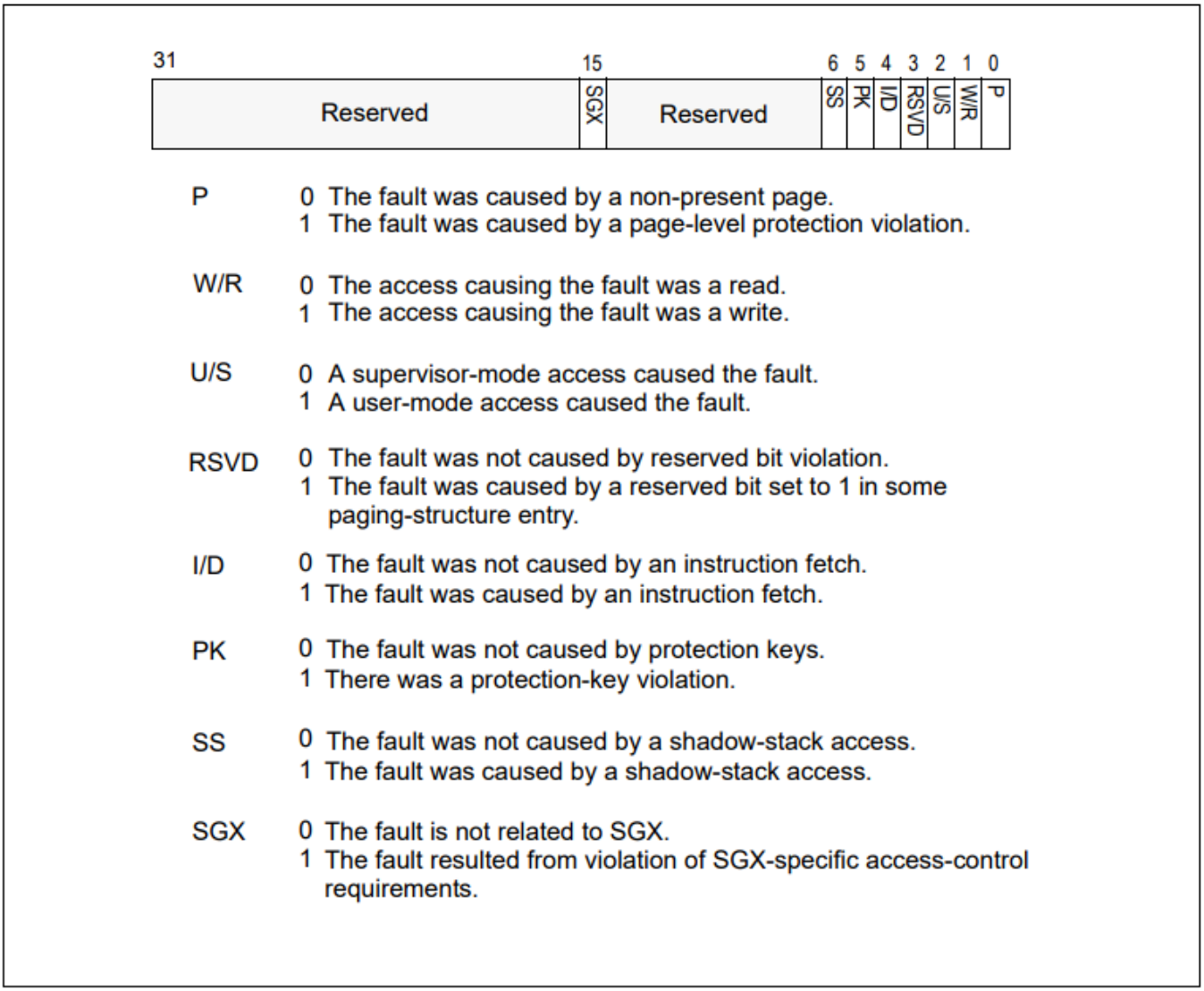


Figure 4-12. Page-Fault Error Code

这块之所以讲这么详细，因为我想让大家知道一切的原理都有最一手资料的来源，这些一手资料写的都非常详细和友好，大家完全不必道听途说，也不必毫无头绪地搜索网上的博客。

当然，与本文相关的，就是这个**存在位 P**。

当触发这个 Page-Fault 中断后，就会进入 Linux 0.11 源码中的 **page_fault** 方法，由于 Linux 0.11 的 page_fault 是汇编写的，很不直观，这里我选 Linux 1.0 的代码给大家看，逻辑是一样的。

```
void do_page_fault(..., unsigned long error_code) {
    ...
    if (error_code & 1)
        do_wp_page(error_code, address, current, user_esp);
    else
        do_no_page(error_code, address, current, user_esp);
    ...
}
```

根据 **error_code** 的不同，有不同的逻辑。

刚刚说了，这个中断是由于 **0x8000000** 这个线性地址的访问，遇到了页表项的**存在位 P** 等于 0 的情况，所以 error_code 的第 0 位就是 0，会走 **do_no_page** 逻辑。

之前在讲 第30回 | 番外篇 - 写时复制就这么几行代码 的时候，讲了 **do_wp_page**，这是在 P=1 时的逻辑，文章的结尾我说过，后面会把页表项的存在位 P 为 0 时触发的 do_no_page 逻辑讲给大家，这不就来了么。

do_wp_page 叫**页写保护中断**，do_no_page 叫**缺页中断**。

好了，我们用了很大篇幅，说明白了跳转到一个 P=0 的地址会发生什么，接下来就是具体看 do_no_page 函数的逻辑咯。

缺页中断 do_no_page

我们先一睹为快它的代码。

```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long error_code,unsigned long address) {
    int nr[4];
    unsigned long tmp;
    unsigned long page;
    int block,i;

    address &= 0xfffff000;
    tmp = address - current->start_code;
    if (!current->executable || tmp >= current->end_data) {
        get_empty_page(address);
        return;
    }
    if (share_page(tmp))
        return;
    if (!(page = get_free_page()))
        oom();
/* remember that 1 block is used for header */
    block = 1 + tmp/BLOCK_SIZE;
    for (i=0 ; i<4 ; block++,i++)
        nr[i] = bmap(current->executable,block);
    bread_page(page,current->executable->i_dev,nr);
    i = tmp + 4096 - current->end_data;
    tmp = page + 4096;
    while (i-- > 0) {
        tmp--;
        *(char *)tmp = 0;
    }
    if (put_page(page,address))
        return;
    free_page(page);
    oom();
}
```

我们仍然是去掉一些不重要的分支，假设跳转不会超过数据末端 end_data，也没有共享内存页面，申请空闲内存时也不会内存不足产生 oom 等，将程序简化如下。

```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long address) {
    // 线性地址的页面地址 0x8000000
    address &= 0xfffff000;
    // 计算相对于进程基址的偏移 0
    unsigned long tmp = address - current->start_code;
    // 寻找空闲的一页内存
    unsigned long page = get_free_page();
    // 计算这个地址在文件中的哪个数据块 1
    int block = 1 + tmp/BLOCK_SIZE;
    // 一个数据块 1024 字节，所以一页内存需要读 4 个数据块
    int nr[4];
    for (int i=0 ; i<4 ; block++,i++)
        nr[i] = bmap(current->executable,block);
    bread_page(page,current->executable->i_dev,nr);
    ...
    // 完成页表的映射
    put_page(page,address);
}
```

这就简单多了，我们还是一点点看。

首先，缺页产生的线性地址，之前假设过了，是 0x8000000，也就是进程 2 自己线性地址空间的起始处 128M 这个位置。

由于我们的页表映射是以**页**为单位的，所以首先计算出 address 所在的页，其实就是完成一次 **4KB 的对齐**。

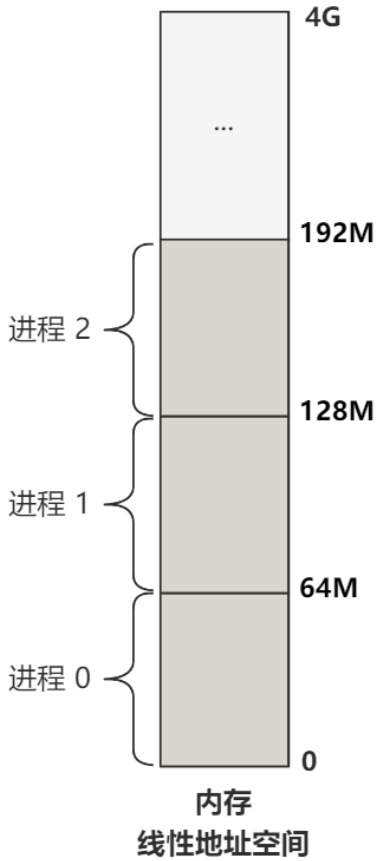
```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long address) {
    // 线性地址的页面地址 0x8000000
    address &= 0xfffff000;
    ...
}
```

此时 address 对齐后仍然是 0x8000000。

这个地址是整个线性地址空间的地址，但对于进程 2 自己来说，需要计算出相对于进程 2 的偏移地址，也就是去掉进程 2 的段基址部分。

```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long address) {
    ...
    // 计算相对于进程基址的偏移 0
    unsigned long tmp = address - current->start_code;
    ...
}
```

这里的 current->start_code 就是进程 2 的段基址，也是 128M。



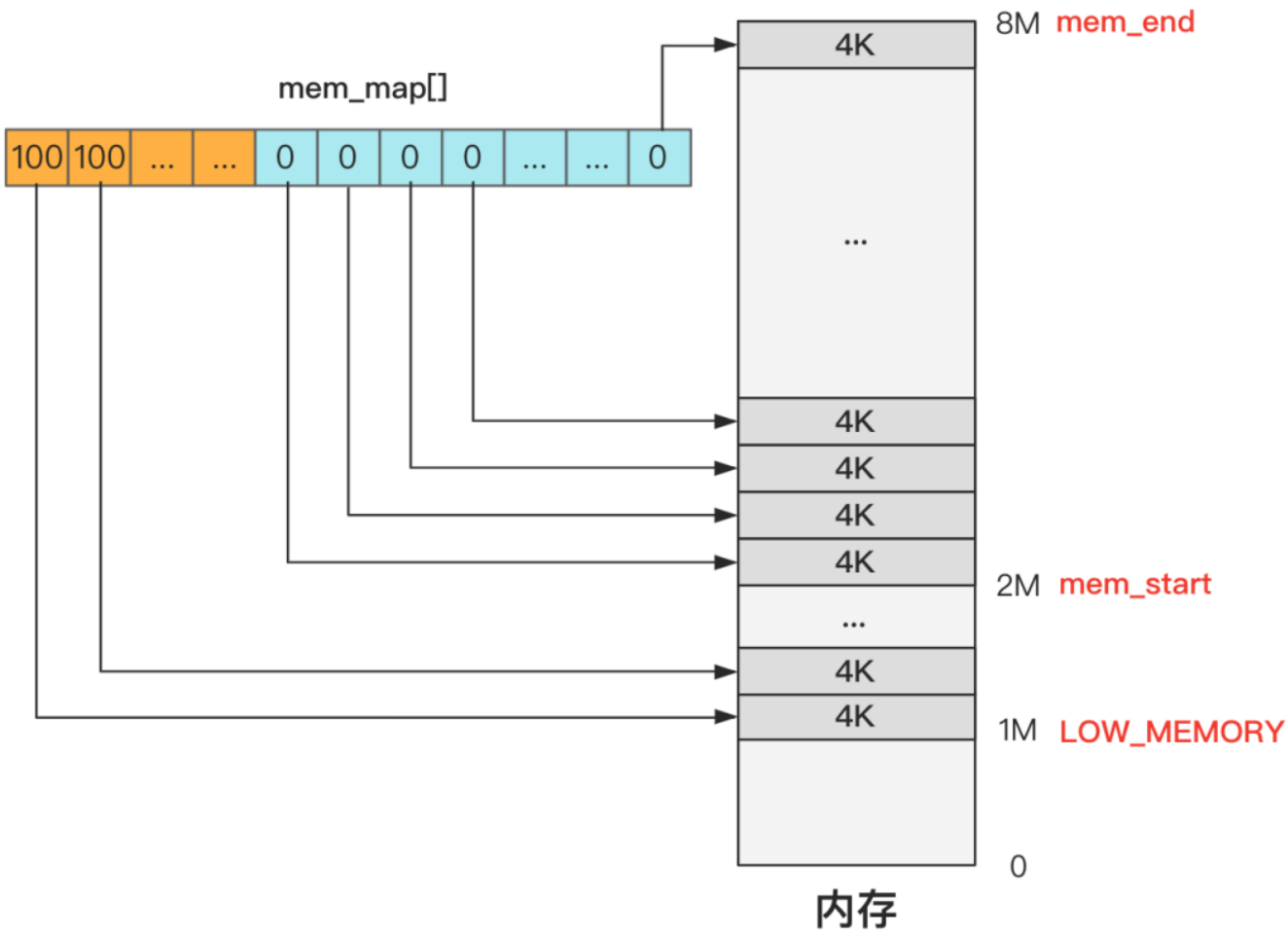
所以偏移地址 tmp 计算后等于 **0**，这和我们之前假设的 a_entry = 0 是一致的。

接下来很简单，就是寻找一个空闲页。

```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long address) {
    ...
    // 寻找空闲的一页内存
    unsigned long page = get_free_page();
    ...
}
```

这个 **get_free_page** 是用汇编语言写的，其实就是去 **mem_map[]** 中寻找一个值为 0 的位置，这就表示找到了空闲内存。

这部分忘记的同学，可以看一下 第13回 | 主内存初始化 mem_init，之前苦苦建立的一些初始化的数据结构，就用上了。



找到一页物理内存后，当然是把硬盘中的数据加载进来，下面的代码就是完成这个工作。

```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long address) {
    ...
    // 计算这个地址在文件中的哪个数据块 1
    int block = 1 + tmp/BLOCK_SIZE;
    // 一个数据块 1024 字节，所以一页内存需要读 4 个数据块
    int nr[4];
    for (int i=0 ; i<4 ; block++,i++)
        nr[i] = bmap(current->executable,block);
    bread_page(page,current->executable->i_dev,nr);
    ...
}
```

从硬盘的哪个位置开始读呢? 首先 0 内存地址，应该就对应着这个文件 0 号数据块，当然由于 /bin/sh 这个 a.out 格式的文件使用了 1 个数据块作为头部 exec 结构，所以我们**跳过头部**，从文件 1 号数据块开始读。

读多少块呢? 因为硬盘中的 1 个数据块为 1024 字节，而一页内存为 4096 字节，所以要读 4 块，这就是 nr[4] 的缘故。

之后读取数据主要是两个函数，**bmap** 负责将相对于文件的数据块转换为相对于整个硬盘的数据块，比如这个文件的第 1 块数据，可能对应在整个硬盘的第 24 块的位置。

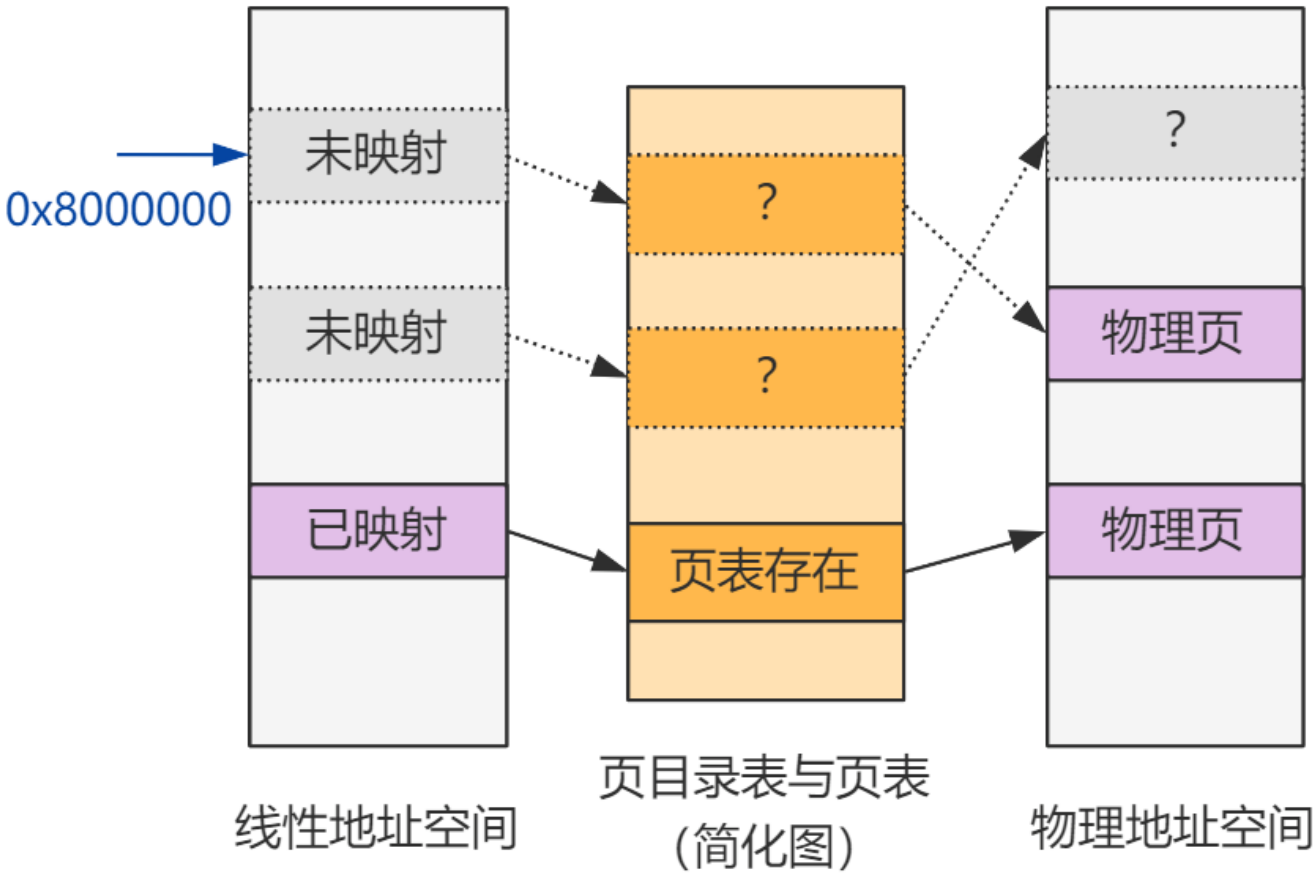
bread_page 就是连续读取 4 个数据块到 1 页内存的函数，这个函数原理就复杂了，之后第五部分会讲这块的内容，但站在用户层的效果很好理解，就是把硬盘数据复制到内存罢了。

OK，现在硬盘上所需要的内容已经被读入物理内存了。

最后一步完成**页表的映射**。

```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long address) {
    ...
    // 完成页表的映射
    put_page(page,address);
}
```

这是因为我们此时仅仅是申请了物理内存页，并且把硬盘数据复制了进来，但我们并没有把这个物理内存页和线性地址空间的内存页进行映射，也就是没建立相关的**页表**。



建立页表的映射，由于 Linux 0.11 使用的是二级页表，所以实际上就是写入**页目录项**和**页表项**的过程，我把 **put_page** 函数简化了一下，只考虑页目录项还不存在的场景。

```
// memory.c
unsigned long put_page(unsigned long page,unsigned long address) {
    unsigned long tmp, *page_table;
    // 找到页目录项
    page_table = (unsigned long *) ((address>>20) & 0xffc);
    // 写入页目录项
    tmp = get_free_page();
    *page_table = tmp|7;
    // 写入页表项
    page_table = (unsigned long *) tmp;
    page_table[(address>>12) & 0x3ff] = page | 7;
    return page;
}
```

大家可以结合页目录表和页表的数据结构看一下，很简单，就是个计算过程。

关于页目录表和页表这些**分页**相关的知识，可以回顾之前的 第9回 | Intel 内存管理两板斧：分段与分页，这里就不再赘述。

缺页中断返回

好了，这就是整个缺页中断处理的过程，本质上就是加载硬盘对应位置的数据，然后建立页表的过程。

```
// memory.c
// address 缺页产生的线性地址 0x8000000
void do_no_page(unsigned long address) {
    // 线性地址的页面地址 0x8000000
    address &= 0xfffff000;
    // 计算相对于进程基址的偏移 0
    unsigned long tmp = address - current->start_code;
    // 寻找空闲的一页内存
    unsigned long page = get_free_page();
    // 计算这个地址在文件中的哪个数据块 1
    int block = 1 + tmp/BLOCK_SIZE;
    // 一个数据块 1024 字节，所以一页内存需要读 4 个数据块
    int nr[4];
    for (int i=0 ; i<4 ; block++,i++)
        nr[i] = bmap(current->executable,block);
    bread_page(page,current->executable->i_dev,nr);
    ...
    // 完成页表的映射
    put_page(page,address);
}
```

再回过头看整个代码，是不是清晰了不少？

好，那我们再往上看，我们之前是在进程 2 里执行了 execve 函数将程序替换成 /bin/sh，也就是 shell 程序。

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

execve 函数返回后，CPU 就跳转到 /bin/sh 程序的第一行开始执行，但由于跳转到的线性地址不存在，所以引发了今天我们讲的**缺页中断**，把硬盘里 /bin/sh 所需要的内容加载到了内存，此时缺页中断返回。

返回后，CPU 会再次尝试跳转到 0x8000000 这个线性地址，此时由于缺页中断的处理结果，**使得该线性地址已有对应的页表进行映射**，所以顺利地映射到了物理地址，也就是 /bin/sh 的代码部分（从硬盘加载过来的），那接下来就终于可以执行 /bin/sh 程序，也就是 shell 程序了。

那这个 shell 程序到底是啥呢？他的代码并不在 Linux 0.11 的源码里，所以我们的重点将不是分析它的源码，仅仅了解它的原理即可。

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。

收录于合集 #操作系统源码 43

上一篇
调试 Linux 最早期的代码

下一篇
第37回 | shell 程序跑起来了

forcats | tidyverse家族对「分类变量」的解决方案（上）

R语言学堂

外部函数如何访问其它类的私有成员

程序喵大人