写时复制就这么几行代码，麻烦你先看看再 BB 行吗？

Original 闪客 低并发编程 2022-03-20 17:30

收录于合集
#操作系统源码
43个

这里讲的是 Linux 内核里的写时复制原理。

写时复制的原理网上讲述的文章很多，今天来一篇很直接的文章，通过看看 Linux 0.11 这个最简单的操作系统，从源码层面把写时复制的原理搞清楚。

很简单哦，你可别中途就放弃了。

直接干！

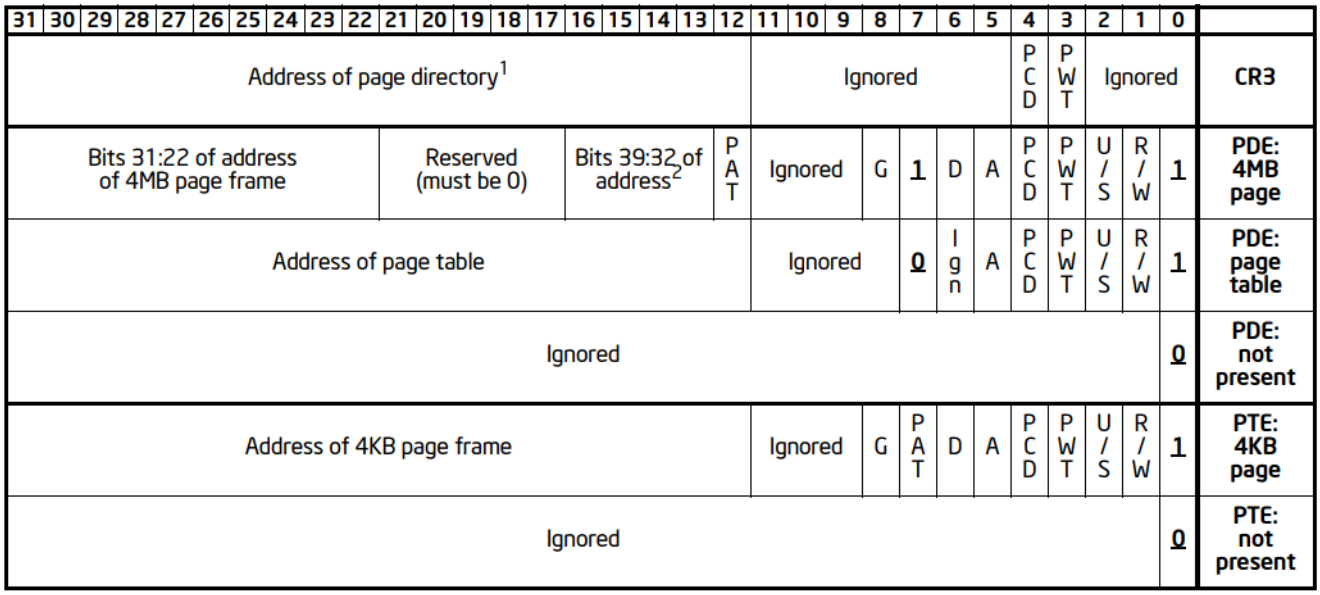
哦不行，干之前先来点储备知识，如果你已经有了这一 pa 可以略过，不过我估计你没有...

储备知识

坚持看完这部分，写时复制用到的这里的知识点只有其中一个位的值而已，但我把周边也给你讲讲。

32 位模式下，Intel 设计了**页目录表**和**页表**两种结构，用来给程序员们提供分页机制。

在 Intel Volume-3 Chapter 4.3 Figure 4-4 中给出了页表和页目录表的数据结构，PDE 就是页目录表，PTE 就是页表。



31–12	11–9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]	Ignored					PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame; Reserved (must be 0); Bits 39:32 of address[2]; PAT	Ignored	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table	Ignored		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored										0	PDE: not present
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.

大部分的操作系统使用的都是 4KB 的页框大小，Linux 0.11 也是，所以我们只看 4KB 页大小时的情况即可。

一个由程序员给出的逻辑地址，**要先经过分段机制的转化变成线性地址，再经过分页机制的转化变成物理地址。**

Figure 4-2 给出了线性地址到物理地址，也就是**分页机制**的转化过程。

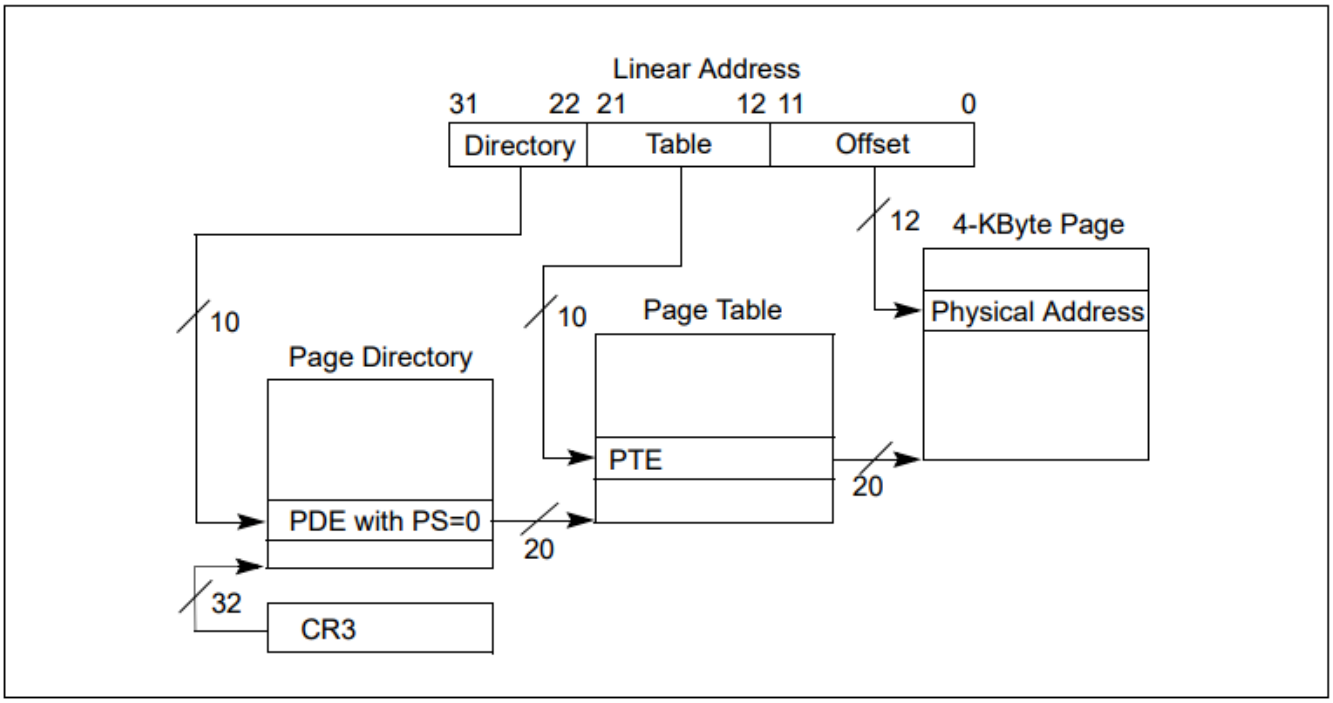


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

这里的 PDE 就是页目录表，PTE 就是页表，刚刚说过了。

在手册接下来的 Table 4-5 和 Table 4-6 中，详细解释了页目录表和页表数据结构各字段的含义。

Table 4-5 是页目录表。

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

Table 4-6 是页表。

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

他们几乎都是一样的含义，我们就只看页表就好了，看一些比较重要的位。

31:12 表示页的起始物理地址，加上线性地址的后 12 位偏移地址，就构成了最终要访问的内

存的物理地址，这个就不说了。

第 0 位是 P，表示 Present，存在位。

第 1 位是 RW，表示读写权限，0 表示只读，那么此时往这个页表示的内存范围内写数据，则不允许。

第 2 位是 US，表示用户态还是内核态，0 表示内核态，那么此时用户态的程序往这个内存范围内写数据，则不允许。

在 Linux 0.11 的 head.s 里，初次为页表设置的值如下。

```
setup_paging:
   ...
    movl $pg0+7,_pg_dir     /* set present bit/user r/w */
    movl $pg1+7,_pg_dir+4       /*  --------- " " --------- */
    movl $pg2+7,_pg_dir+8       /*  --------- " " --------- */
    movl $pg3+7,_pg_dir+12      /*  --------- " " --------- */
    movl $pg3+4092,%edi
    movl $0xfff007,%eax     /*  16Mb - 4096 + 7 (r/w user,p) */
    std
1:  stosl
    ...
```

后三位是 7，用二进制表示就是 111，即初始设置的 4 个页目录表和 1024 个页表，都是：

存在（1），可读写（1），用户态（1）

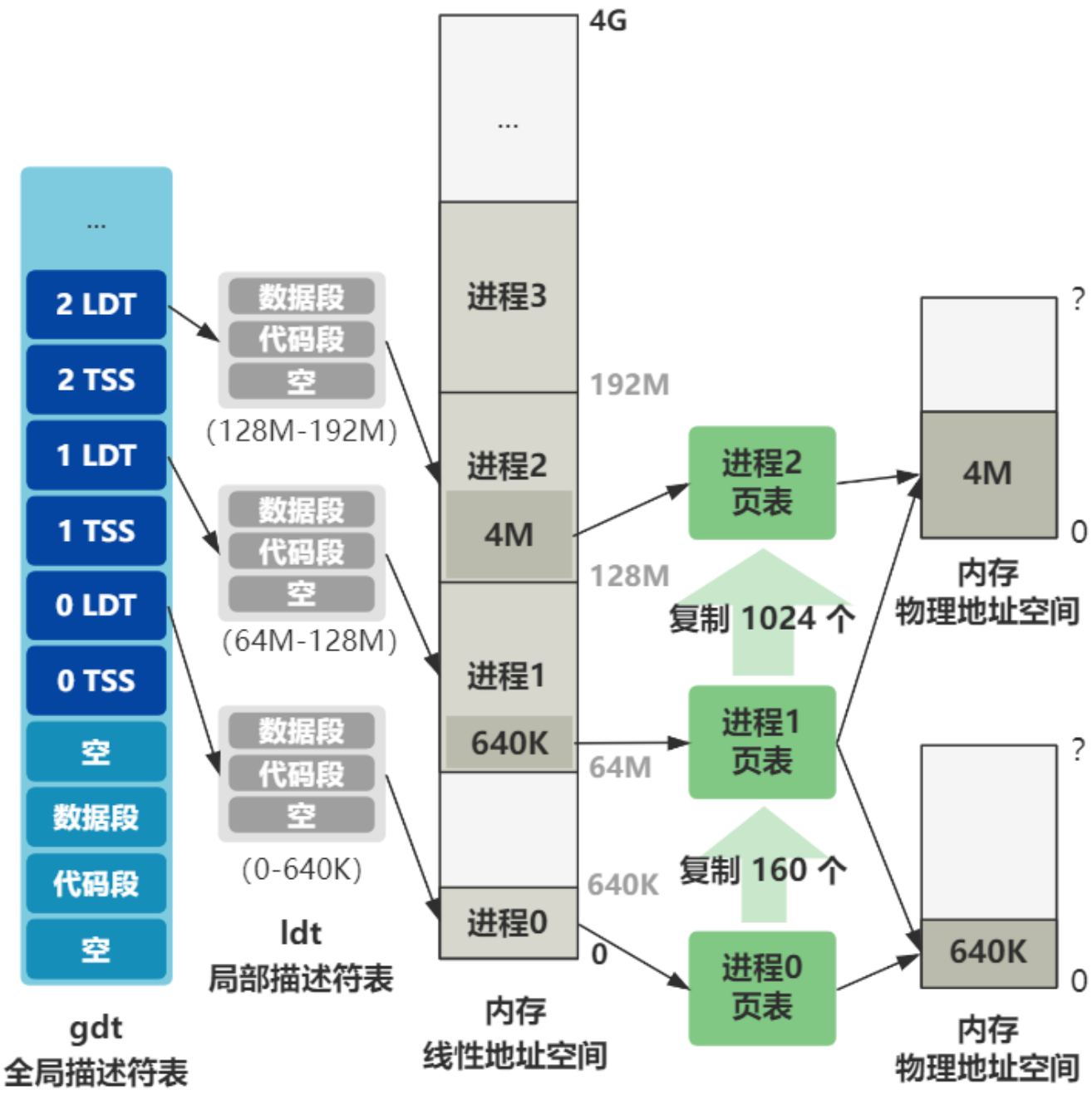
好了，储备知识就到这里。

如果你前面没读懂，你只需要知道，页表当中有一位是表示读\写的，而 Linux 0.11 初始化时，把它设置为了 1，表示可读写。

写时复制的本质

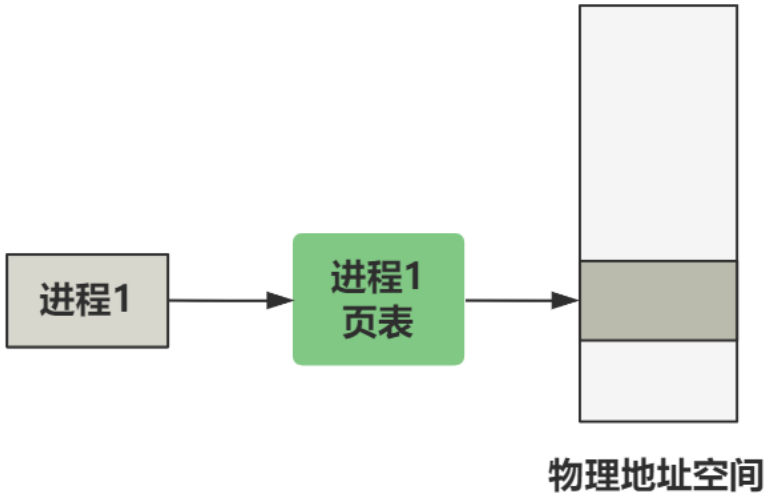
在调用 fork() 生成新进程时，新进程与原进程会共享同一内存区。只有当其中一个进程进行写操作时，系统才会为其另外分配内存页面。

之前在我的操作系统系列，我给过一个 Linux 0.11 进程的内存规划图。

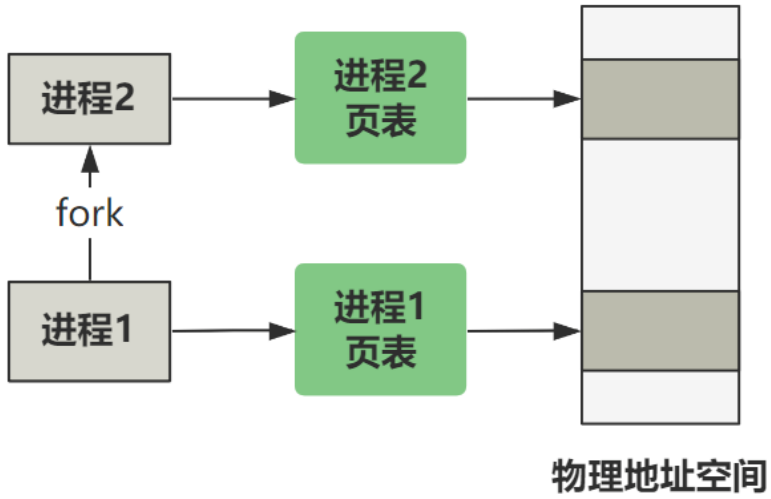


不过我们考虑写时复制并不用这么复杂，去掉些细节就是。

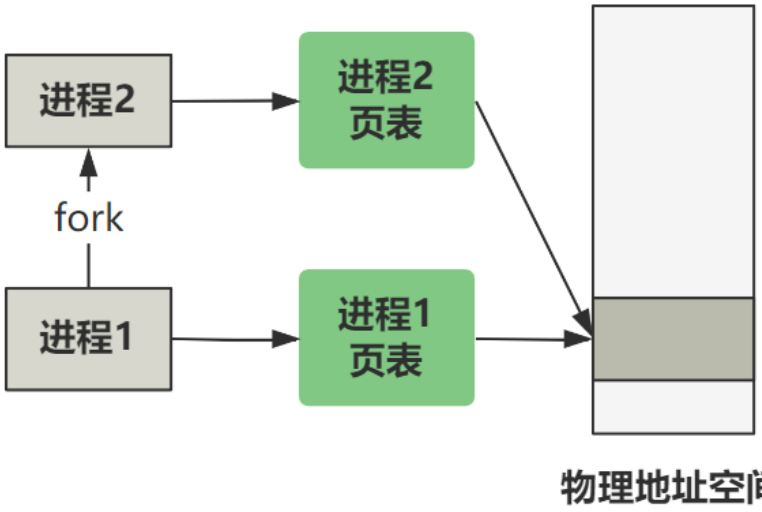
原来的进程通过自己的页表占用了一定范围的物理内存空间。



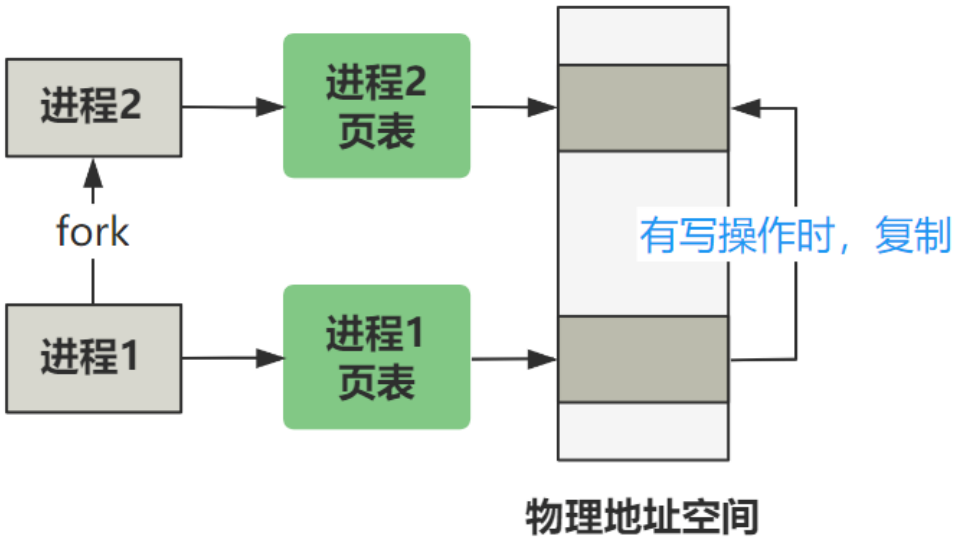
调用 fork 创建新进程时，原本页表和物理地址空间里的内容，都要进行复制，因为进程的内存空间是要隔离的嘛。



但 fork 函数认为，复制物理地址空间里的内容，比较费时，**所以姑且先只复制页表，物理地址空间的内容先不复制。**



如果只有读操作，那就完全没有影响，复不复制物理地址空间里的内容就无所谓了，这就很赚。**但如果有写操作，那就不得不把物理地址空间里的值复制一份**，保证进程间的内存隔离。



有写操作时，再复制物理内存，就叫**写时复制。**

看看代码咋写的

有上述的现象，必然是在 fork 时，对**页表**做了手脚，这回知道为啥储备知识里讲页表结构了吧？

同时，只要有写操作，就会触发写时复制这个逻辑，这是咋做到的呢？答案是通过**中断**，具体

是**缺页中断**。

好的，首先来看 fork。

fork 细节很多，具体可以看 一个新进程的诞生（六）fork 中进程基本信息的复制 和 一个新进程的诞生（七）透过 fork 来看进程的内存规划，这里只看其中关键的复制页表的代码。

```
int copy_page_tables(...) {
    ...
    // 源页表和新页表一样
    this_page = *from_page_table;
    ...
    // 源页表和新页表均置为只读
    this_page &= ~2;
    *from_page_table = this_page;
    ...
}
```

还记得知识储备当中的页表结构吧，就是把 **R/W** 位置 **0** 了。

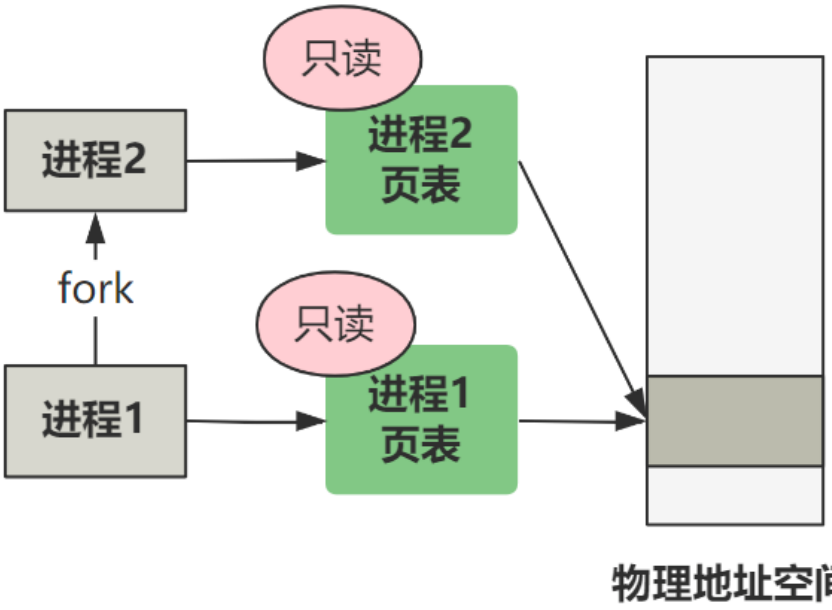
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)								Bits 39:32 of address[2]		PAT	Ignored		G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.

用刚刚的 fork 图表示就是。



那么此时，再次对这块物理地址空间进行写操作时，就不允许了。

但不允许并不是真的不允许，Intel 会触发一个**缺页中断**，具体是 **0x14** 号中断，中断处理程序里边怎么处理，那就由 Linux 源码自由发挥了。

Linux 0.11 的缺页中断处理函数的开头是用汇编写的，看着太闹心了，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);
    ...
}
```

可以看出，根据中断异常码 **error_code** 的不同，有不同的逻辑。

那触发缺页中断的异常码都有哪些呢？

在 Intel Volume-3 Chapter 4.7 Figure 4-12 中给出。

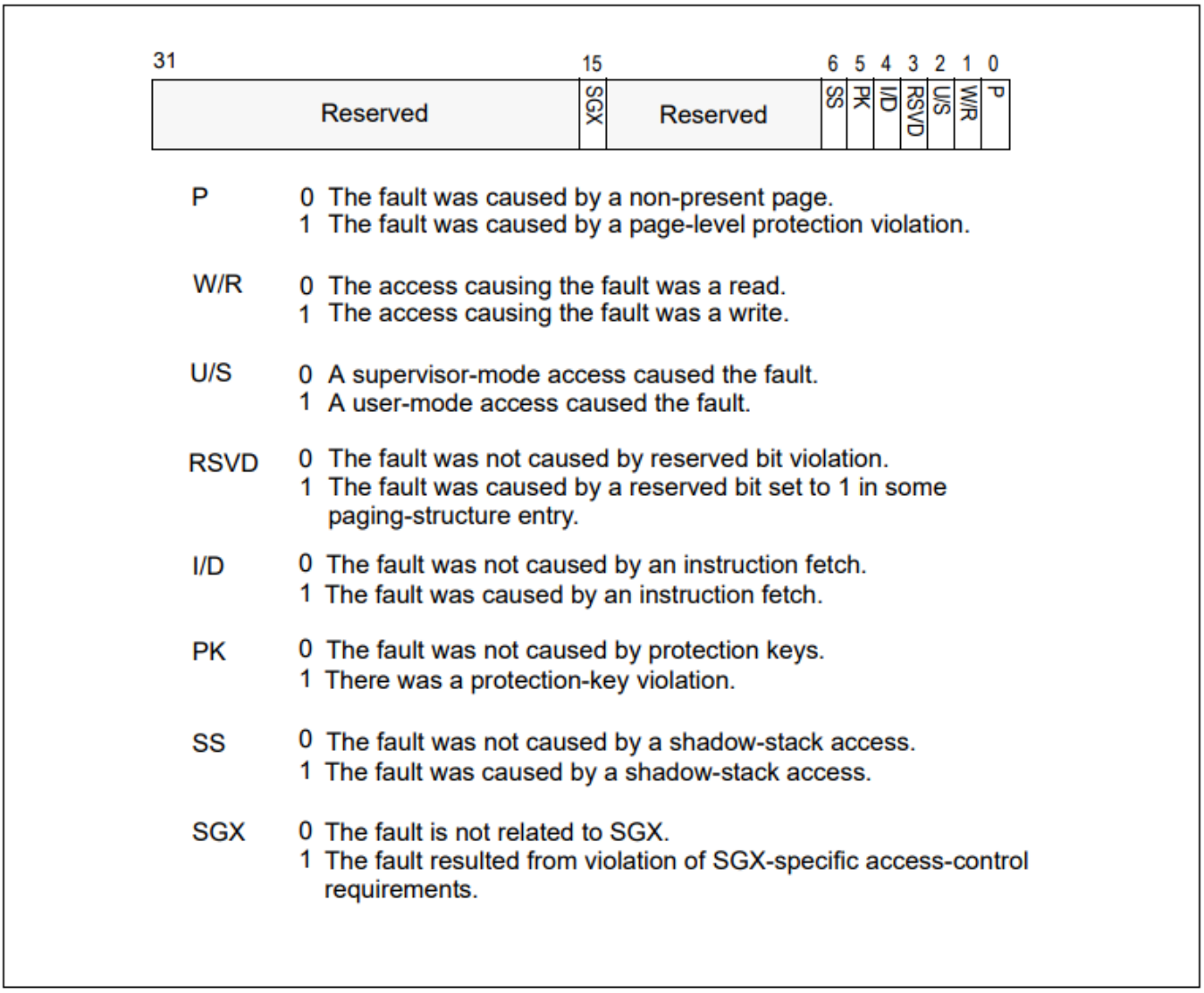


Figure 4-12. Page-Fault Error Code

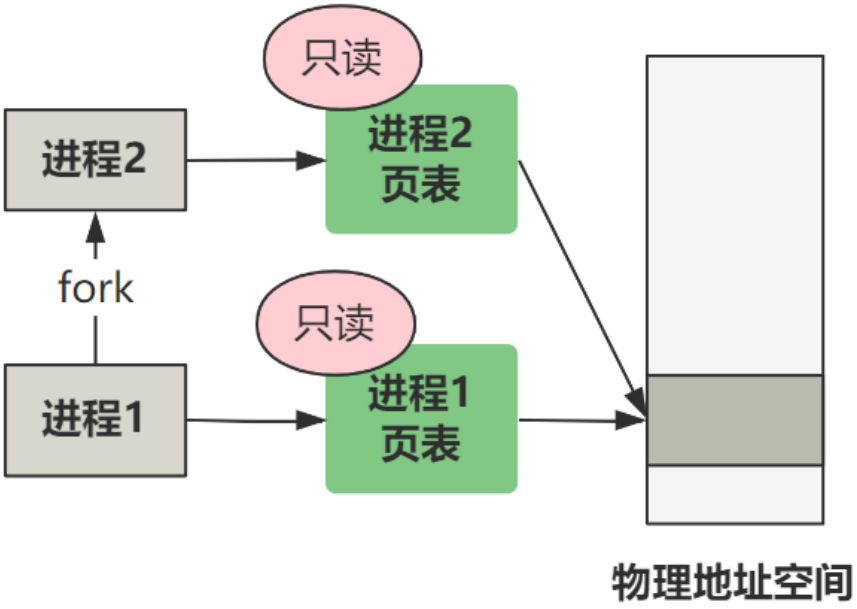
可以看出，当 error_code 的第 0 位，也就是存在位为 0 时，会走 do_no_page 逻辑，其余情况，均走 do_wp_page 逻辑。

我们 fork 的时候只是将读写位变成了只读，存在位仍然是 1 没有动，所以会走 **do_wp_page** 逻辑。

```
void do_wp_page(unsigned long error_code,unsigned long address) {
    // 后面这一大坨计算了 address 在页表项的指针
    un_wp_page((unsigned long *)
        (((address>>10) & 0xffc) + (0xfffff000 &
        *((unsigned long *) ((address>>20) &0xffc)))));
}

void un_wp_page(unsigned long * table_entry) {
    unsigned long old_page,new_page;
    old_page = 0xfffff000 & *table_entry;
    // 只被引用一次，说明没有被共享，那只改下读写属性就行了
    if (mem_map[MAP_NR(old_page)]==1) {
        *table_entry |= 2;
        invalidate();
        return;
    }
    // 被引用多次，就需要复制页表了

    new_page=get_free_page();
    mem_map[MAP_NR(old_page)]--;
    *table_entry = new_page | 7;
    invalidate();
    copy_page(old_page,new_page);
}

// 刷新页变换高速缓冲宏函数
#define invalidate() \
__asm__("movl %%eax,%%cr3"::"a" (0))
```

我用图直接说明这段代码的细节。

刚刚 fork 完一个进程，是这个样子的对吧?



这是我们对着这个物理空间范围，写一个值，就会触发上述函数。

假如是进程 2 写的。

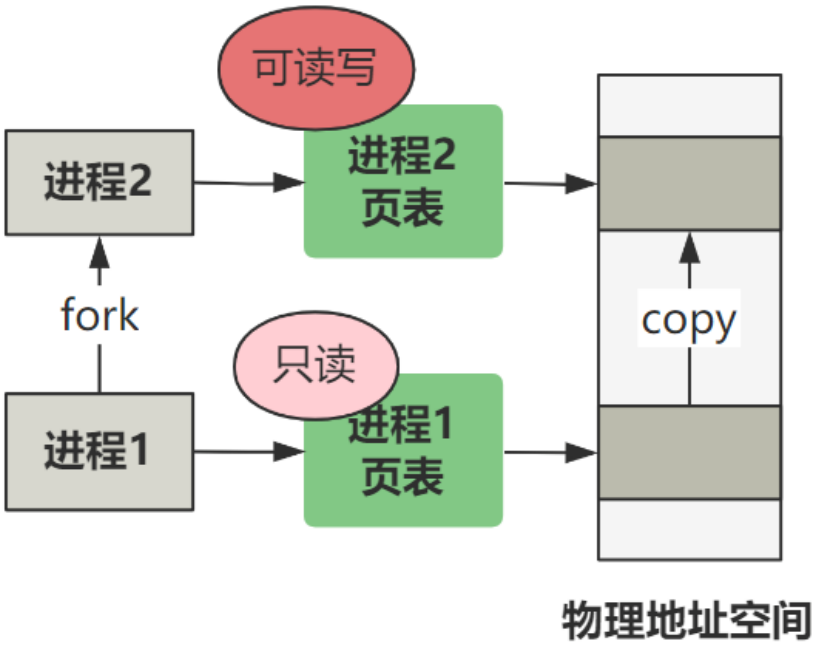
显然此时这个物理空间被引用了大于 1 次，所以要复制页面。

```
new_page=get_free_page();
```

并且更改页面只读属性为可读写。

```
*table_entry = new_page | 7;
```

图示就是这样。

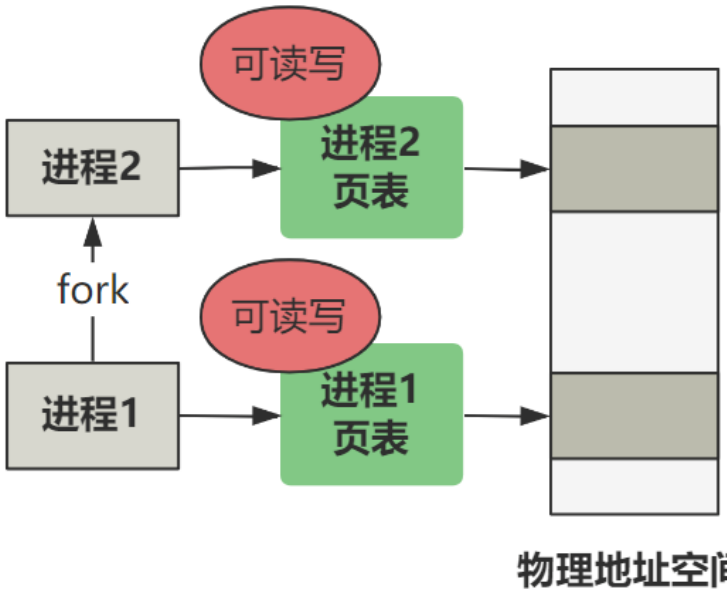


是不是很简单。

那此时如果进程 1 再写呢？那么引用次数就等于 1 了，只需要更改下页属性即可，不用进行页面复制操作。

```
if (mem_map[MAP_NR(old_page)]==1) ...
```

图示就是这样。



就这么简单。

是不是从细节上看，和你原来理解的写时复制，还有点不同。

缺页中断的处理过程中，除了写时复制原理的 do_wp_page，还有个 **do_no_page**，是在页表项的存在位 P 为 0 时触发的。

这个和**进程按需加载内存**有关，如果还没加载到内存，会通过这个函数将磁盘中的数据复制到内存来，这个有时间再给大家讲。

如果你对类似的这些知识想有系统性的了解，每次都单看这种一篇一篇的技术散文是无效的，有系统性的了解后再读这些文章，你会收获很大。

如何有系统性的了解呢？可以尝试追更我现在写的操作系统系列吧。目前已经完整更完三大部分了，还正在持续更新中。

第一部分 进入内核前的苦力活

开篇词
第一回 | 最开始的两行代码
第二回 | 自己给自己挪个地儿
第三回 | 做好最最基础的准备工作
第四回 | 把自己在硬盘里的其他部分也放到内存来
第五回 | 进入保护模式前的最后一次折腾内存
第六回 | 先解决段寄存器的历史包袱问题
第七回 | 六行代码就进入了保护模式
第八回 | 烦死了又要重新设置一遍 idt 和 gdt
第九回 | Intel 内存管理两板斧：分段与分页
第十回 | 进入 main 函数前的最后一跃！
第一部分完结 进入内核前的苦力活

第二部分 大战前期的初始化工作

第11回 | 整个操作系统就 20 几行代码
第12回 | 管理内存前先划分出三个边界值
第13回 | 主内存初始化 mem_init
第14回 | 中断初始化 trap_init
第15回 | 块设备请求项初始化 blk_dev_init
第16回 | 控制台初始化 tty_init

第17回 | 时间初始化 time_init
第18回 | 进程调度初始化 sched_init
第19回 | 缓冲区初始化 buffer_init
第20回 | 硬盘初始化 hd_init
第二部分总结与回顾

第三部分：一个新进程的诞生

第21回 | 新进程诞生全局概述
第22回 | 从内核态切换到用户态
第23回 | 如果让你来设计进程调度
第24回 | 从一次定时器滴答来看进程调度
第25回 | 通过 fork 看一次系统调用
第26回 | fork 中进程基本信息的复制
第27回 | 透过 fork 来看进程的内存规划
第三部分总结与回顾

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
一个新进程的诞生 完结撒花！！！

下一篇
第31回 | 拿到硬盘信息

Modified on 2022-03-20

Read more

设计原型与代码的实现

探探设计