10 | 页中断：fork、mmap背后的保护神

2021-11-15 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 22:51 大小 20.93M

你好，我是海纳。

这节课是对前面所有课程的一次总结和回顾。前面我们介绍了很多内存管理的相关机制，其实都是为了把这节课的故事讲完整。在前面的课程里，我们了解了进程内部的分布，但也留下了三个关键的问题没有讲清楚：

1. fork 的工作方式非常奇怪，一方面父进程和子进程还可以访问共有的变量，另一方面，它们又可以各自修改这个变量，且这个修改对方都看不见，这是怎么做到的呢？
2. 我们在第 1 节课讲内存映射时，就讲过页表中未映射状态的页表项，并不存在一块具体的物理内存与之对应。但是当我们访问到这一页的时候，页表项可以自动变成已映射的正常状态。谁在背后做了什么事情呢？

3. mmap 的功能十分强大，这些强大的能力是怎么完成的呢？

这三个问题，虽然看上去相互之间关系不大，但实际上它们背后都依赖**页中断机制**。

页中断和普通的中断一样，它的中断服务程序入口也在 IDT 中（第 2 节课的内容），但它是由 MMU 产生的硬件中断。**页中断有两类重要的类型：写保护中断和缺页中断。正是这两类中断在整个系统的后台默默地工作着，就像守护神一样支撑着内存系统正常工作。**

大多数时候，我们即使不知道它们的存在，程序也能正常地运行。但是有时候，程序写得不好就有可能造成中断频繁发生，从而带来巨大的性能下降。面对这种情况，我们第一时间就应该想到统计页中断。因为除了页中断本身会带来性能下降之外，统计页中断也可以反推程序的运行特点，从而为进一步分析程序瓶颈点，提供数据和思路。

讲到这，我想你应该意识到掌握页中断的必要性了，其实这也是我们这节课的学习目标，同时我们还将借此解决上面提到的三个问题。好，不啰嗦了，我们先了解下页中断有哪些类型吧。

页中断有哪些类型？

在之前的课程里，我们介绍了页表映射的原理，也提到过页表项里定义了页的读写属性等等。如果物理页不在内存中，或者页表未映射，或者读写请求不满足页表项内的权限定义时，MMU 单元就会产生一次中断。

我们在第 2 节课中详细介绍了中断机制和 IDT 的结构，并且在介绍中断向量时提到过页中断的向量是 14。所以，操作系统在启动以后，它会把处理页中断的程序入口地址，设置到 IDT 的 14 号中断描述符里。在 Linux 系统上，页中断服务程序的名称是 do_page_fault。

当中断发生以后，CPU 会自动地在栈里存放一个错误码，来区分页中断的类型，还会把发生页中断的虚拟地址放到 CR2 寄存器，这样，中断服务程序就可以清楚地知道是什么原因导致的中断，然后才能做出相应的处理。

根据中断来源的不同，页中断大致可以分为以下几种类型：

异常来源	中断服务程序要做的动作
页面未映射	分配物理页面，并且在页表中设置好从虚拟地址到物理地址的映射
页面内容在磁盘中	如果物理页面被交换出去，或者文件映射页表尚未加载，服务程序就会从磁盘中将数据按页读入内存
写只读页面	如果是写时复制页，就复制一页，标记为可写分配给进程；如果是写异常，就触发SIGSEGV
没有访问权限	给进程发送SIGSEGV信号

极客时间

从这个表格里，你会发现，页中断服务程序根据不同的情况，兢兢业业地为整个系统的内存管理，默默做着贡献。接下来，我们就带着这节课开头提出的三个问题，来看看页中断是怎么工作的。我们先从第一个问题，fork 的原理是什么开始吧。

fork 原理：写保护中断与写时复制

我们前面说，父进程和子进程不仅可以访问共有的变量，还可以各自修改这个变量，并且这个修改对方都看不见。这其实是 fork 的一种写时复制机制，这一点我们在第 5 节课中模糊提到过，而里面起关键作用的就是写保护中断。下面我们来看看这到底是怎么一回事。

实际上，操作系统为每个进程提供了一个进程管理的结构，在偏理论的书籍里一般会称它为进程控制块（Process Control Block，PCB)。具体到 Linux 系统上，PCB 就是 task_struct 这个结构体。它里面记录了进程的页表基址，打开文件列表、信号、时间片、调度参数和线性空间已经分配的内存区域等等数据。

其中，**描述线性空间已分配的内存区域的结构对于内存管理至关重要**，我们先来看一下这个结构。在 Linux 源码中，负责这个功能的结构是 vm_area_struct，后面简称 vma。内

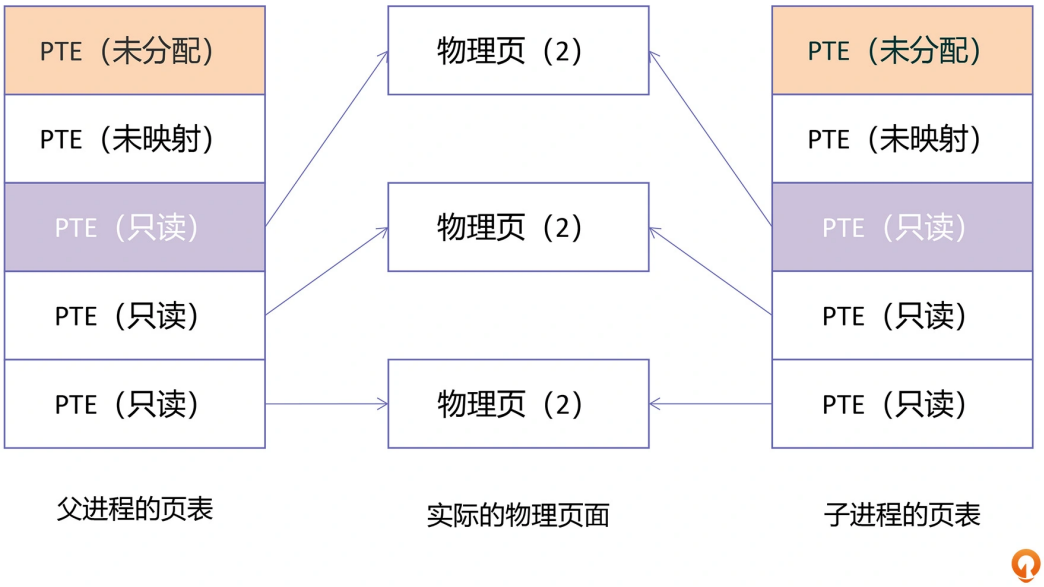
核将每一段具有相同属性的内存区域当作一个单独的内存对象进行管理。vma 中比较重要的属性我列在下面：

复制代码

```
struct vm_area_struct {
  unsigned long vm_start;      // 区间首地址
  unsigned long vm_end;        // 区间尾地址
    pgprot_t      vm_page_prot;  // 访问控制权限
    unsigned long vm_flags;      // 标志位
    struct file * vm_file;       // 被映射的文件
    unsigned long vm_pgoff;      // 文件中的偏移量
  ...
}
```

在操作系统内核里，fork 的第一个动作是把 PCB 复制一份，但类似于物理页等进程资源不会被复制。这样的话，父进程与子进程的代码段、数据段、堆和栈都是相同的，这是因为它们拥有相同的页表，自然也有相同的虚拟空间布局和对物理内存的映射。如果父进程在 fork 子进程之前创建了一个变量，打开了一个文件，那么父子进程都能看到这个变量和文件。

fork 的第二个动作是复制页表和 PCB 中的 vma 数组，并把所有当前正常状态的数据段、堆和栈空间的虚拟内存页，设置为不可写，然后把已经映射的物理页面的引用计数加 1。这一步只需要复制页表和修改 PTE 中的写权限位可以了，并不会真的为子进程的所有内存空间分配物理页面，修改映射，所以它的效率是非常高的。这时，父子进程的页表的情况如下图所示：



在上图中，物理页括号中的数字代表该页被多少个进程所引用。Linux 中用于管理物理页面，和维护物理页的引用计数的结构是 mem_map 和 page struct。

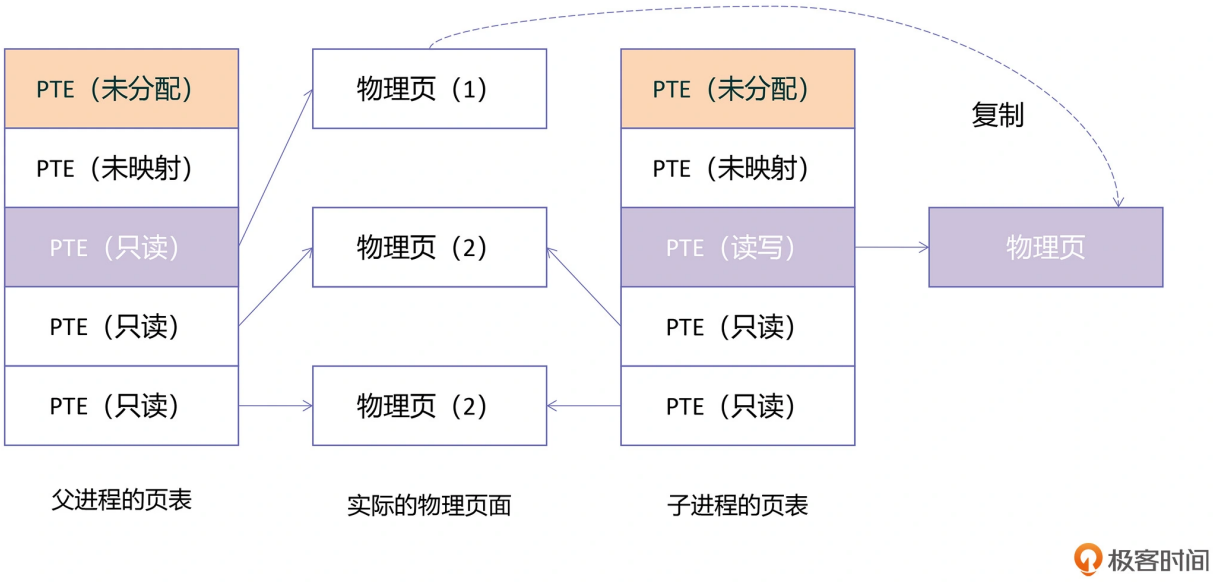
这两个动作执行完后，fork 调用就结束了。此时，由于有父进程和子进程两个 PCB，操作系统就会把两个进程都加入到调度队列中。当父进程得到执行，它的 IP 寄存器还是指向 fork 调用中，所以它会从这个调用中返回，只不过返回值是子进程的 PID。当子进程得到执行时，它的 IP 寄存器也是停在 fork 调用中，它从这个调用中返回，其返回值是 0。

接下来，就是写保护中断要发挥作用的地方了。不管是父进程还是子进程，它们接下来都有可能发生写操作，但我们知道在 fork 的第二步操作中，已经将所有原来可写的地方都变成不可写了，所以这时必然会发生写保护中断。

我们刚才说，Linux 系统的页中断的入口地址是 do_page_fault，在这个函数里，它会继续判断中断的类型。由于发生中断的虚拟地址在 vma 中是可写的，在 PTE 中却是只读的，可以断定这是一次写保护中断。这时候，内核就会转而调用 do_wp_page 来处理这次中断，wp 是 write protection 的缩写。

在 do_wp_page 中，系统会首先判断发生中断的虚拟地址所对应的物理地址的引用计数，如果大于 1，就说明现在存在多个进程共享这一块物理页面，那么它就需要为发生中断的进程再分配一个物理页面，把老的页面内容拷贝进这个新的物理页，最后把发生中断的虚

拟地址映射到新的物理页。这就完成了一次写时复制 (Copy On Write， COW）。具体过程如下图所示：



在上图中，当子进程发生写保护中断后，系统就会为它分配新的物理页，然后复制页面，再修改页表映射。这时老的物理页的引用计数就变为 1，同时子进程中的 PTE 的权限也从只读变为读写。

当父进程再访问到这个地址时，也会触发一次写保护中断，这时系统发现物理页的引用计数为 1，那就只要把父进程 PTE 中的权限，简单地从只读变为读写就可以了。这个过程比较简单，我就不画图了，你可以自己思考一下。

fork 之后如果要执行新的程序，那么就需要执行 execve 这个系统调用。它的主要作用是加载可执行程序并运行。接下来我们就看看这个函数背后的故事。

execve 原理：缺页中断

接着来说这节课开始时所提到的第二个问题，未映射页面是如何自动变成正常页面的？我们将通过 execve 的例子来进行分析。

execve 的作用是使当前进程执行一个新的可执行程序，它的原型如下所示：

复制代码

```
1
```

```
#include <unistd.h>

int execve(const char* filename, const char* argv[],
           const char* envp[])
```

其中 execve 的第一个参数是可执行程序的文件名，第二个参数用来传递命令行参数，第三个参数用来传递环境变量。

execve 的执行步骤如下所示：

1. 清空页表，这样整个进程中的页都变成不存在了，一旦访问这些页，就会发生页中断；
2. 打开待加载执行的文件，在内核中创建代表这个文件的 struct file 结构；
3. 加载和解析文件头，文件头里描述了这个可执行文件一共有多少 section；
4. 创建相应的 vma 来描述代码段，数据段，并且将文件的各个 section 与这些内存区域建立映射关系；
5. 如果当前加载的文件还依赖其他共享库文件，则找到这个共享库文件，并跳转到第 2 步继续处理这个共享库文件；
6. 最后跳转到可执行程序的入口处执行。

我们在第 3 节课讲了 section 与内存中的 segment 的对应关系。**execve 的实现并不负责将文件内容加载到物理页中，它只建立了这种文件 section，与内存区域的映射关系就结束了。**真正负责加载文件内容的是缺页中断，接下来，我们就看看缺页中断是如何加载物理页的。

在 execve 的执行步骤中，我们讲了，内核为可执行程序创建一个 vma 结构体实例，然后将它的 vm_file 属性设成第 2 步所打开的文件，这就建立起了内存区域和文件的映射关系。这个内核区域的区间首地址、区间尾地址和控制权限，都是由第 3 步解析的信息决定的。例如.text 段被加载到的内存首地址，也就是链接时所决定的起始地址，它就决定了内存代码段的起始地址。

由于第 1 步把页表都清空了，这就导致 CPU 在加载指令时会发现代码段是缺失的，此时就会产生缺页中断。

Linux 内核用于处理缺页中断的函数是 do_no_page，如果内核检查，当前出现缺页中断的虚拟地址所在的内存区域 vma（虚拟地址落在该内存区域的 vm_start 和 vm_end 之间）存在文件映射 (vm_file 不为空），那就可以通过虚拟内存地址计算文件中的偏移，这就定位到了内存所缺的页对应到文件的哪一段。然后内核就启动磁盘 IO，将对应的页从磁盘加载进内存。一次缺页中断就这样被解决了。

到这里，第二个问题的答案你就都搞清楚了。**可执行程序的加载不是一次性完成的，而是由缺页中断根据需要，将文件的内容以页为单位加载进内存的，一次只会加载一页。**

搞清楚了 execve 背后的原理，我们再来分析 mmap 的原理，你就很容易理解了，因为它背后的机制仍然是围绕着 vm_area_struct 这个核心结构，由页中断来完成各种功能。

mmap 强大的能力是怎么来的？

在回答这节课开始提出的第三个问题，也就是 mmap 的功能十分强大，这些强大的能力是怎么完成的前，我们先回顾下第 3 节课的内容，mmap 根据映射的类型，有四种最常用的组合：

私有匿名映射，用于分配堆空间；

共享匿名映射，用于父子进程之间通讯；

私有文件映射，用于加载动态链接库；

共享文件映射，用于多进程之间通讯。

我们接下来针对这四种情况依次进行分析。

私有匿名映射

私有匿名映射是最简单的情况，**在调用 mmap 时，只需要在文件映射区域分配一块内存，然后创建这块内存所对应的 vma 结构，这次调用就结束了。**

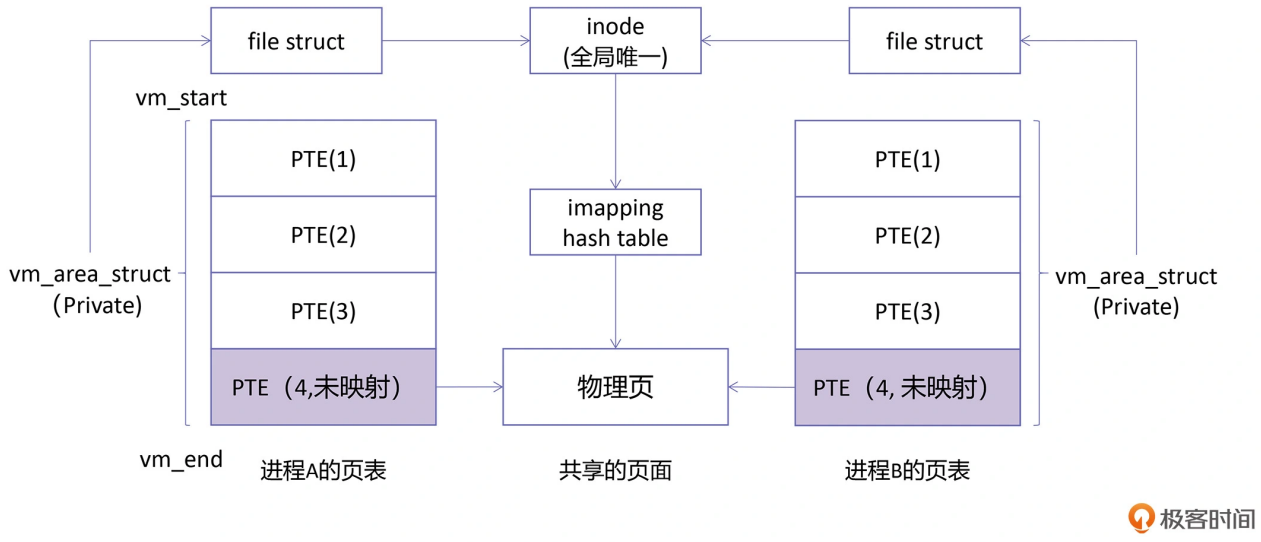
当访问到这块虚拟内存时，由于这块虚拟内存都没有映射到物理内存上，就会发生缺页中断，但这一次的缺页中断与 execve 时的缺页中断不一样，这次是匿名映射，所以关联文件属性为空。此时，内核就会调用 do_anonymous_page 来分配一个物理内存，并将整个物理页全部初始化为 0，然后在页表里建立起虚拟地址到物理地址的映射关系。

私有文件映射

在内核中，如果有一个进程打开了一个文件，PCB 中就会有一个 struct file 结构与这个文件对应。struct file 结构是与进程相关，假如进程 A 与进程 B 都打开了文件 f，那么进程 A 中就会有一个 struct file 结构，进程 B 中也会有一个。

Linux 的文件系统中有一个叫做 inode 的结构，这个结构与具体的磁盘上的文件是一一对应的，也就是说对于同一个文件，整个内核中只会有一个 inode 结构。所以进程 A 与进程 B 的 file struct 结构都有一个指针指向 inode 结构，这就将 file struct 与 inode 结构联系起来了。

在 inode 结构中，有一个哈希表，以文件的页号为 key，以物理内存页为 value。当进程 A 打开了文件 f，然后读取了它的第 4 页，这时，内核就会把 4 和这个物理页，放入这个哈希表中。当进程 B 再打开文件 f，要读取它的第 4 页时，因为 f 的第 4 页的内容已经被加载到物理页中了，所以就不用再加载一次了。只需要将 B 的虚拟地址与这个物理页建立映射就可以了，如下图所示：



我要提醒你的是，**哈希表在现代的 Linux 内核中，已经被优化成了 Radix tree 和最小堆的一种优化的数据结构，它们比哈希表有更好的时间效率，所以你在阅读不同版本的 Linux 内核代码时要注意这个变化。**

如果文件是只读的话，那这个文件在物理页的层面上其实是共享的。也就是进程 A 和进程 B 都有一页虚拟内存被映射到了相同的物理页上。但如果要写文件的时候，因为这一段内

存区域的属性是私有的，所以内核就会做一次写时复制，为写文件的进程单独地创建一份副本。这样，一个进程在写文件时，并不会影响到其他进程的读。

对于共享库文件，代码段的私有属性其实并不影响它在所有进程间共享；但如果数据段在执行的过程发生变化，内核就可以通过写时复制机制为每个进程创建一个副本。这就是对于共享库文件要选择私有文件映射的根本原因。

这里我们就有这样一个结论：**私有文件映射的只读页是多进程间共享的，可写页是每个进程都有一个独立的副本，创建副本的时机仍然是写时复制**。

共享文件映射

在私有文件映射的基础上，共享文件映射就很简单了：**对于可写的页面，在写的时候不进行复制就可以了**。这样的话，无论何时，也无论是读还是写，多个进程在访问同一个文件的同一个页时，访问的都是相同的物理页面。

共享匿名映射

在这节课之前，你可能会觉得共享匿名映射在父子进程间通讯是最简单的，因为父子进程共享了相同的 mmap 的返回值，看上去最直观。但实际上，从内核的角度说，它却是最复杂的。

原因是 **mmap 并不真正分配物理内存，它只是分配了一段虚拟内存，也就是说只在 PCB 中创建了一个 vma 结构而已。这就导致 fork 在复制页表的时候，页表中共享匿名映射区域都是未映射状态**。

请你设想一下，如果内核不做特殊处理的话，在父进程因为访问共享内存区域而遇到缺页中断时，内核为它分配了物理页面，等子进程再访问共享内存区域时，内核也没有办法知道子进程的虚拟内存，应该映射到哪个物理页面上，因为缺页中断只能知道当前进程是谁，以及发生问题的虚拟地址是什么，这些信息不足够计算出，是否有其他进程已经把共享内存准备好了。

在内核中使用虚拟文件系统来解决这个问题之前，早期的 Linux 内核中并不支持共享匿名映射。虚拟文件并不是真实地在磁盘上存在的。它只是由内核模拟出来的，但是它也有自

己的 inode 结构。这样一来，内核就能在创建共享匿名映射区域时，创建一个虚拟文件，并将这个文件与映射区域的 vma 关联起来。

当 fork 创建子进程时，子进程会复制父进程的全部 vma 信息。接下来发生的事情就和共享文件映射完全一样了，我们就不再重复了。

至此，我们才终于把 mmap 的核心原理分析清楚。第三个问题的答案也就很清楚了：**mmap 的功能之所以十分强大，主是因为操作系统综合使用写保护中断、缺页中断和文件机制来实现 mmap 的各种功能**。

总结

这节课，我们先介绍了页中断产生的原因，大致可以分为缺页中断、写保护中断和非法访问造成的中断等等。

接下来，我们深入地分析了 fork 的原理。fork 在执行时，子进程只会复制父进程的 PCB 和页表，并且把所有页表项都设为只读，这个过程并不会复制真正的物理页。只有当父子进程其中一个对页进行写操作的时候，才会复制一个副本出来。这种机制被称为写时复制。

execve 是一种系统调用，用于加载并运行一个可执行文件。它会打开文件，并做好文件的 section 与内存 segment 的映射，这种映射关系维护在 vm_area_struct 中，然后就清空页表退出执行了。

当指令真正访问到内存的时候，由于页表被清空，这时会产生缺页中断，然后，内核就使用 vma 中的文件映射关系，去磁盘上读取相应的内容，将它放到物理页中，最后建立好虚拟地址到物理地址的映射。这是一种按需加载的机制。

我们分析了 mmap 背后的页中断原理，根据映射的类型，我们还介绍了它常用的 4 种组合和作用。其中：

- **私有匿名映射**，在缺页中断的处理过程，会通过 do_anonymous_page 函数申请一块全零的物理页，并建立虚拟地址到物理页的映射，以达成分配内存的目标；

私有文件映射，则借助文件的 inode 结构共享文件的物理缓存页，当发生写操作时，则会出现写时复制，从而保证每一个进程中都有自己的副本；

共享文件映射，在私有文件映射的基础上，只取消了写时复制，这样一个进程就可以看到其他进程对这个页的修改了；

共享匿名映射，借助了虚拟文件系统。内核在父子进程间，使用自己创建的虚拟文件和共享文件映射，来实现共享匿名映射。

最后，你要特别注意的一点是，Linux 内核为了优化性能，还引入了大量的结构，这使得研究内存管理的源代码变得非常困难。我们这里主要介绍了设计思路，而不会涉及到具体的细节，如果想研究 Linux 内存管理的源码的话，你还可以继续参考《Understanding the Linux Virtual Memory Manager》、《Linux 内核设计与实现》和《深入理解 LINUX 内核》等资料。

通过这节课的学习，我相信我已经帮你建立起了基本的骨架，填充更多细节的任务就交给你自己完成吧。

思考题

我们先将第 3 节课的 mmap 的例子，映射方式修改为 MAP_PRIVATE。请你结合本节课所学的知识分析它的运行结果。欢迎你在留言区分享你的想法和收获，我在留言区等你。

复制代码

```
#include <sys/mman.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid;
    char* shm = (char*)mmap(0, 4096, PROT_READ | PROT_WRITE,
        MAP_PRIVATE| MAP_ANONYMOUS, -1, 0);
    if (!(pid = fork())){
        sleep(1);
        printf("child got a message: %s\n", shm);
        sprintf(shm, "%s", "hello, father.");
        exit(0);
    }
    sprintf(shm, "%s", "hello, my child");
    sleep(2);
    printf("parent got a message: %s\n", shm);
```

```
18      return 0;
19  }
```

吊打面试官

- 谈谈共享库的加载过程?

这道题目与第8节课所讲的动态链接不是一回事，在第8节课，我们关心的是符号是如何解析的，而这里的题目问的是共享库是如何被加载进内存的。

共享库是通过系统调用mmap映射到内存的，映射方式为只读。共享库中的代码段，因为不会被修改，所以多个进程可以共享，其背后的原理是各个进程的页表对共享库文件的代码段映射的物理内存相同。

对于共享库中的全局变量，因为映射方式为只读，所以会对全局变量添加写保护，这样当有进程需要对全局变量修改时，会触发写时复制，在内存中新开辟一个物理页，供这个进程单独使用，通过这样来保证了各个进程全局变量私有。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 09 | 深入理解堆：malloc和内存池是怎么回事？

下一篇 11 | 即时编译：高性能JVM的核心秘密

精选留言 (5)

写留言

linker
2021-11-17

思考题:后执行的进程应该会发生crash

展开

qinsi
2021-11-16

参考资料里的《Linux 内核设计与实现》似乎还是旧版的，新版（第三版）里已经提到了page cache从hash改为radix tree的理由，比如hash的冲突解决使用了链表，对于不存在的key需要遍历完整个链表才能知道不存在；又比如radix tree（又名压缩前缀树）可以压缩存储相同前缀的key，比hash更节省空间。

展开

费城的二鹏
2021-11-16

老师，私有文件映射，多个进程加载同一个文件，其中一个进程进行写入时会发生写时复制。那么，其它进程能否感知到这个文件的变化？

展开

费城的二鹏

2021-11-16

吊打面试官中，共享库加载时设置为只读，代码段与全局变量都是只读的，在写入是会进行复制。系统如何保护代码段不会被写入的？这两种内容的只读标记是是相同的吗？

大鑫仔Yeah

2021-11-15

作者能不能后续讲解下非自动管理内存的语言，比如 Swift