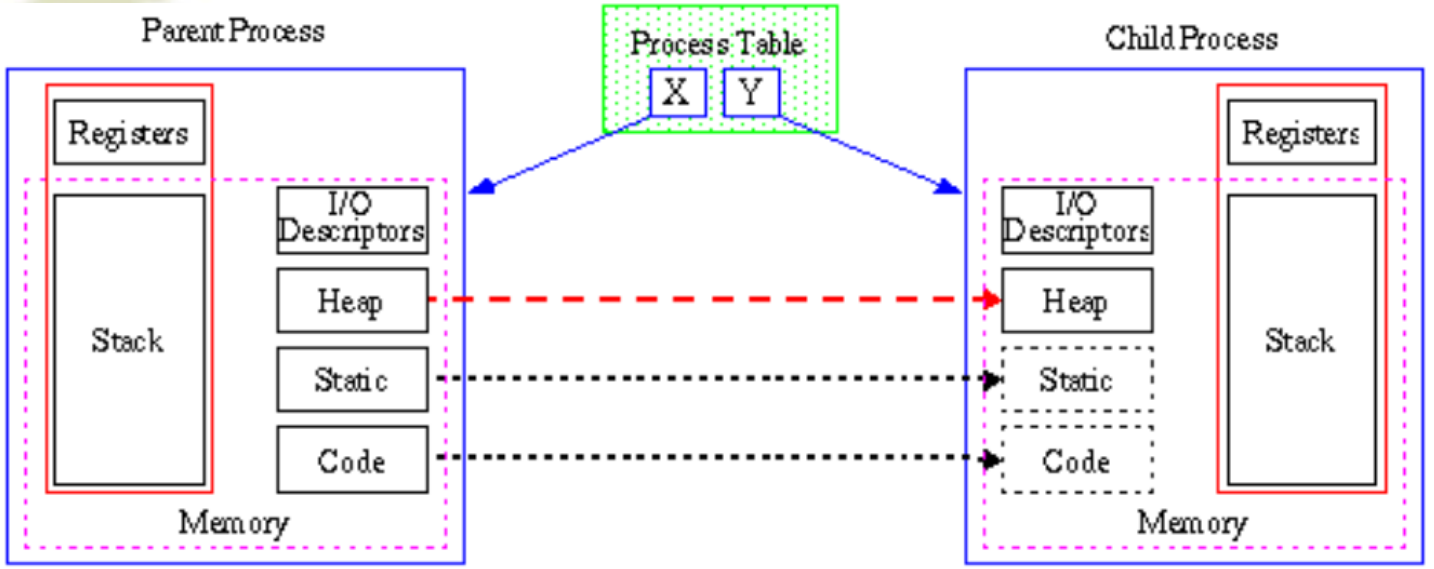
4.2 进程用户空间的创建

西安邮电大学

创建进程用户空间

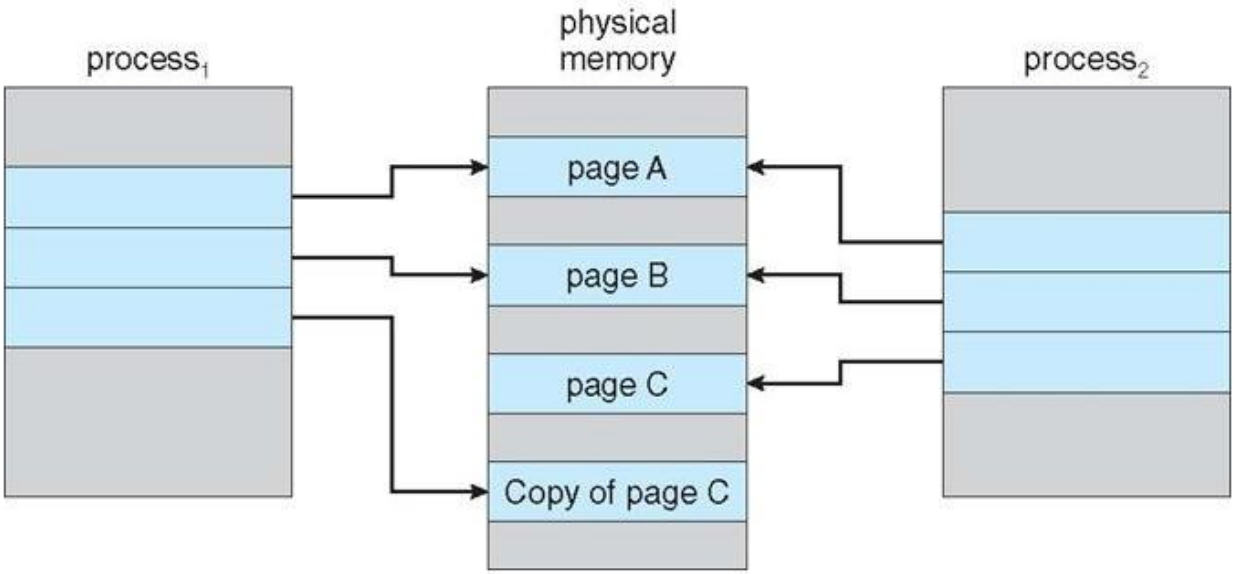
前面我们介绍了每个进程都有自己独立的地址空间，那么，进程的地址空间到底是什么时候创建的？

- 实际上，当fork()系统调用在创建新进程时也为该进程创建完整的用户空间



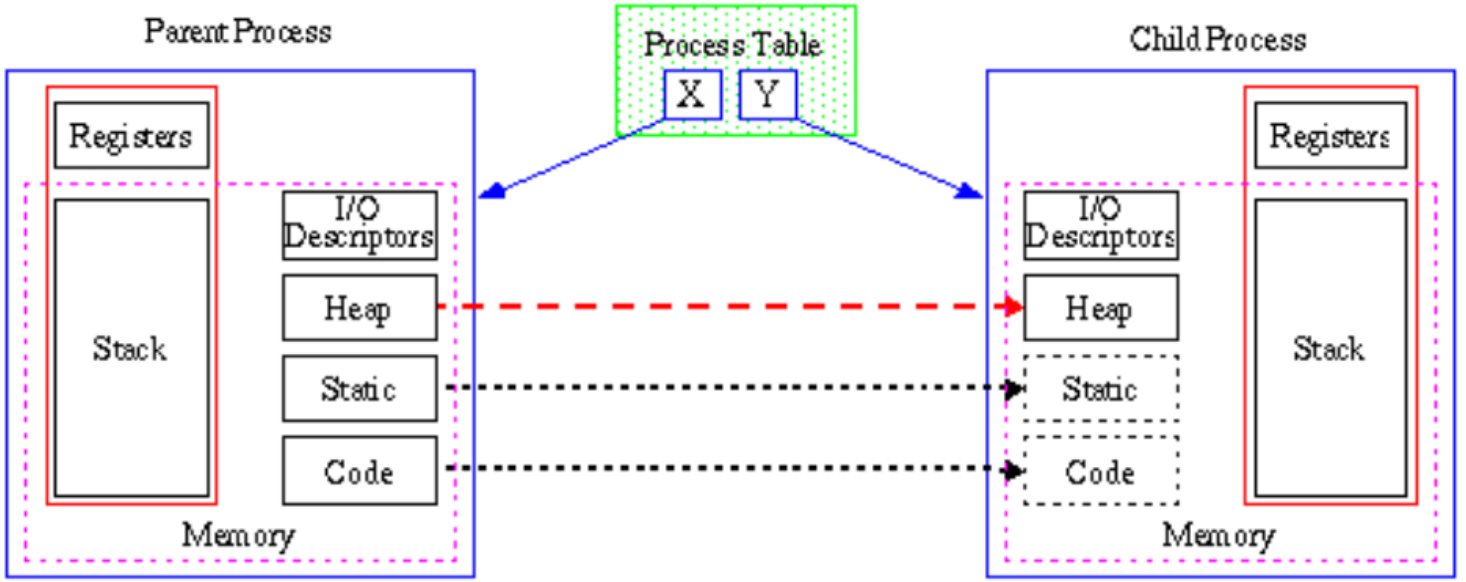
- 那么这个用户空间是如何被创建的，实际上是通过拷贝或共享父进程的用户空间来实现的，即内核调用copy_mm()函数，为新进程建立所有页表和mm_struct结构（通常，每个进程都有自己的用户空间，但是调用clone( )函数创建的内核线程时共享父进程的用户空间）
- Linux利用“写时复制”技术来快速创建进程

写时复制（Copy-on-Write）



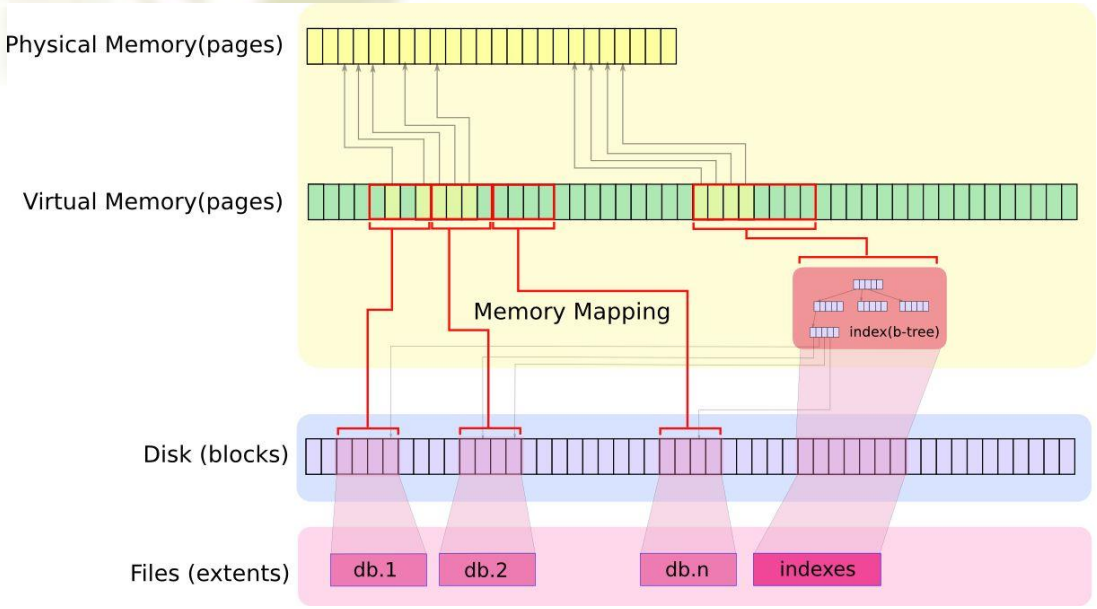
子进程共享父进程的地址空间，只要其中任何一个进程要进行写入，则该页面被复制一份，如图，子进程要写C页，则该页被复制一份。

fork()为什么能快速的创建进程？



从上面的介绍我们可以看出，进程用户空间的创建主要依赖于父进程，而且，在创建的过程中所做的工作仅仅是mm_struc结构的建立、vm_area_struct结构的建立以及页目录和页表的建立，并没有真正地复制一个物理页面，这也是为什么Linux内核能迅速地创建进程的原因之一。

虚存映射

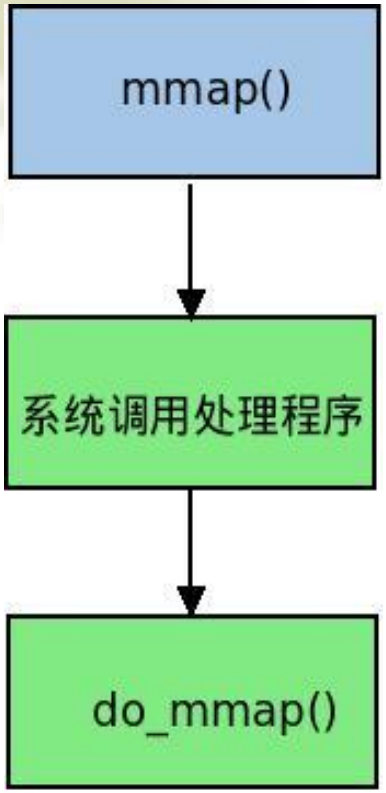


当调用exec()系统调用开始执行一个进程时，进程的可执行映像（包括代码段、数据段，堆和栈等）必须装入到进程的用户地址空间。如果该进程用到了任何一个共享库，则共享库也必须装入到进程的用户空间。

由此可看出，Linux并不将映像装入到物理内存，相反，可执行文件只是被映射到进程的用户空间中。这种将可以执行文件映像映射到进程用户空间的方法被称为“虚存映射”。

那么内核通过怎样的方式新建一个个虚存区（VMA）？

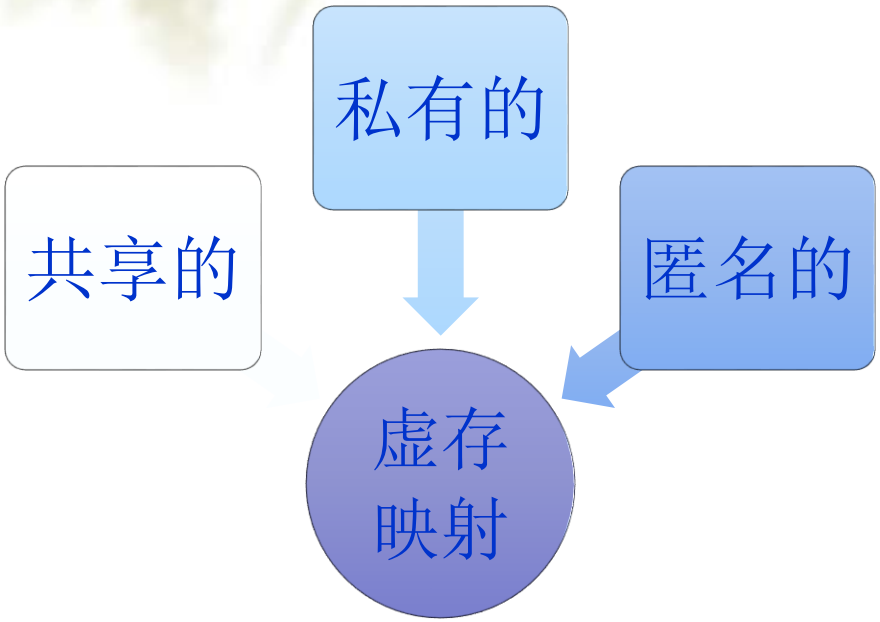
VMA的新建方法



在用户空间可以通过mmap()系统调用获取do_mmap()（指向do_mmap）的功能

在内核空间，可以直接调用do_mmap()创建一个新的虚存区，其调用关系如图：

虚存映射



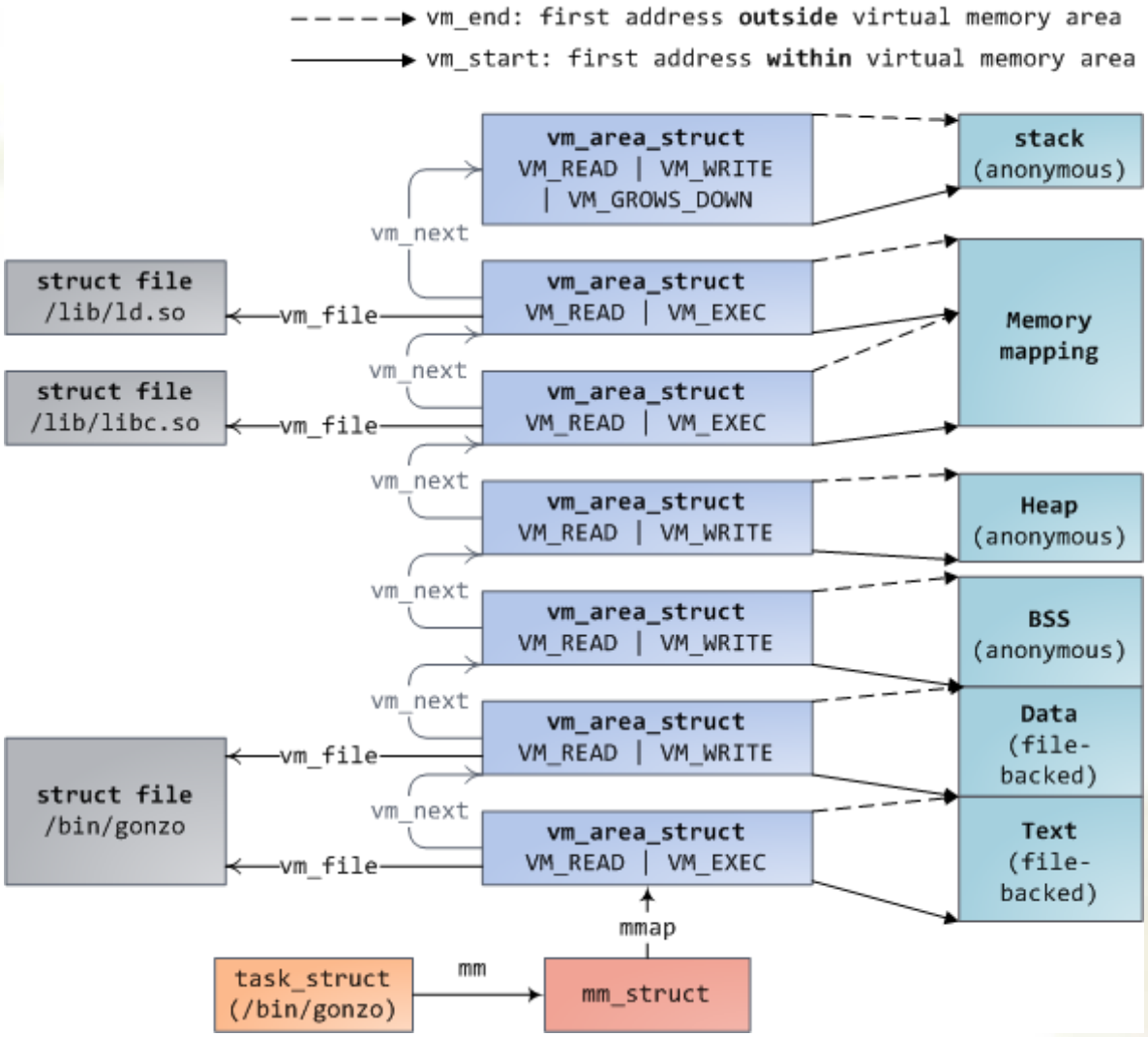
有共享的、私有的虚存映射和匿名映射

（1）共享的：有几个进程共享这一映射，也就是说，如果一个进程对共享的虚存区进行写，其它进程都能感觉到，而且会修改磁盘上对应的文件，文件的共享就可以采用这种方式。

（2）私有的：进程创建的这种映射只是为了读文件，而不是写文件，因此，对虚存区的写操作不会修改磁盘上的文件，由此可以看出，私有映射的效率要比共享映射的高。

除了这两种映射外，如果映射与文件无关，就叫匿名映射。

虚存映射



当可执行映像映射到进程的用户空间时，将产生一组vm_area_struct结构来描述各虚拟区间的起始点和终止点。

进程的虚存区举例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv)
{
        int i;
        unsigned char *buff;
        buff = (char *)malloc(sizeof(char)*1024);
        printf("My pid is :%d\n", getpid());
        for (i = 0; i < 60; i++) {
                sleep(60);
        }
        return 0;
}
```

现在，我们通过一个简单的例子来描述Linux内核是如何把共享库及各个程序段映射到进程的用户空间的。我们考虑一个最简单的C程序exam.c：

进程的虚存区举例

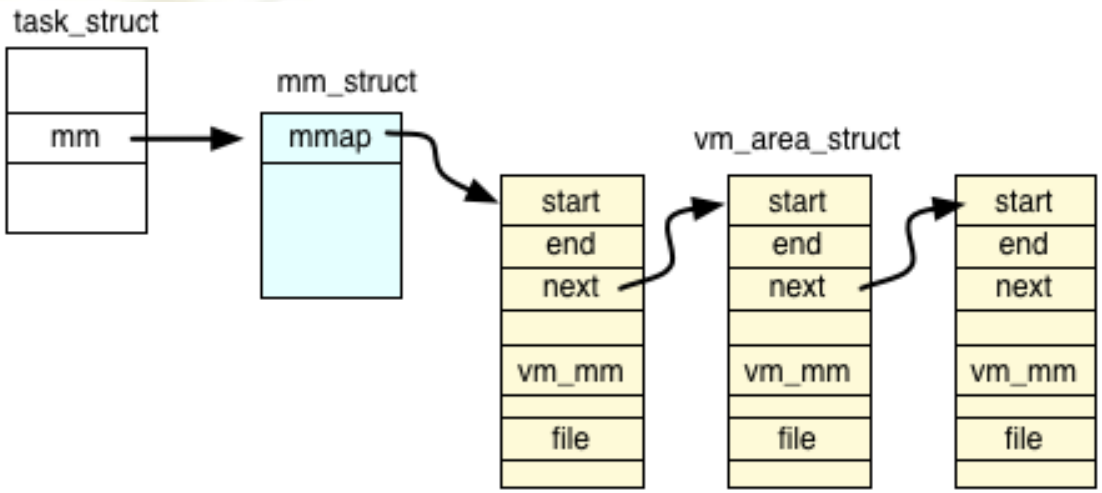
假设程序执行后其对应的PID为9413，这个进程对应的虚存区如表所示，可以从/proc/9413/maps 得到这些信息(注意，这里列出的所有区都是私有虚存映射（在许可权列出现的字母p）。这是因为，这些虚存区的存在仅仅是为了给进程提供数据；当进程执行指令时，可以修改这些虚存区的内容，但与它们相关的磁盘上的文件保持不变，这就是私有虚存映射所起的保护作用。

地址范围	许可权	偏移量	所映射的文件
08048000-08049000	r-xp	00000000	/home/test/exam
08049000-0804a000	rw-p	00001000	/home/test/exam
40000000-40015000	r-xp	00000000	/lib/ld-2.3.2.so
40015000-40016000	rw-p	00015000	/lib/ld-2.3.2.so
40016000-40017000	rw-p	00000000	匿名
4002a000-40159000	r-xp	00000000	/lib/libc-2.3.2.so
40159000-4015e000	rw-p	0012f000	/lib/libc-2.3.2.so
4015e000-40160000	rw-p	00000000	匿名
bfffe000-c0000000	rwxp	fffff000	匿名

与用户空间相关的主要系统调用

系统调用	描述
fork()	创建具有新的用户空间的进程，用户空间中的所有页被标记为“写时复制”，且由父子进程共享。
mmap()	在进程的用户空间内创建一个新的虚存区。
munmap()	销毁一个完整的虚存区或其中的一部分，如果要取消的虚存区位于某个虚存区的中间，则这个虚存区被划分为两个虚存区。
exec()	装入新的可执行文件以代替当前用户空间。
Exit()	销毁进程的用户空间及其所有的虚存区。

编写虚存区内核模块



我们可以编写内核模块，查看某一进程的虚拟区。

在进程的task_struct（PCB）结构中，有如下定义：

```
struct task_struct {
……
    struct mm_struct *mm; /*描述进程的整个用户空间*/
……
  }
```

在stuct mm_struct 结构中，又有如下定义：

```
struct mm_struct{
……
    struct vm_area_struct * mmap; /*描述进程的虚存区*/
……
}
```

编写虚存区内核模块

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/sched.h>

static int pid;

module_param(pid,int,0644);

static int __init memtest_init(void)
{
        struct task_struct *p;
        struct vm_area_struct *temp;

        printk("The virtual memory areas(VMA) are:\n");
        p = pid_task(find_vpid(pid), PIDTYPE_PID); /*
该函数因内核版本而稍有不同*/
        temp = p->mm->mmap;
```

编写一个内核模块，打印进程的虚存区，其中通过模块参数把进程的pid传递给模块。该模块中，通过传递的pid，可以获得对应的PCB，从而可以打印出其各个虚存区的起始地址。

编写虚存区内核模块

```
while(temp) {
        printk("start:%p\tend:%p\n", (unsigned long *)temp->vm_start,
(unsigned long *)temp->vm_end);
        temp = temp->vm_next;
    }

    return 0;
}
static void __exit memtest_exit(void)
{
    printk("Unloading my module.\n");
    return;
}
module_init(memtest_init);
module_exit(memtest_exit);
MODULE_LICENSE("GPL");
```

我们先运行前面写的程序,然后把这个进程的pid作为参数带入模块，如下：

```
$ ./exam &
pid is :9413
$ sudo insmod mem.ko pid=9413
$dmesg
```

内核模块输出信息

地址范围	许可权	偏移量	所映射的文件
08048000-08049000	r-xp	00000000	/home/test/exam
08049000-0804a000	rw-p	00001000	/home/test/exam
40000000-40015000	r-xp	00000000	/lib/ld-2.3.2.so
40015000-40016000	rw-p	00015000	/lib/ld-2.3.2.so
40016000-40017000	rw-p	00000000	匿名
4002a000-40159000	r-xp	00000000	/lib/libc-2.3.2.so
40159000-4015e000	rw-p	0012f000	/lib/libc-2.3.2.so
4015e000-40160000	rw-p	00000000	匿名
bfffe000-c0000000	rwxp	fffff000	匿名

可以看出，输出的信息与前面从proc文件系统中所读取的信息是一致的。

进程的用户空间是由一个个的虚存区组成。对进程用户空间的管理在很大程度上依赖于对虚存区的管理。

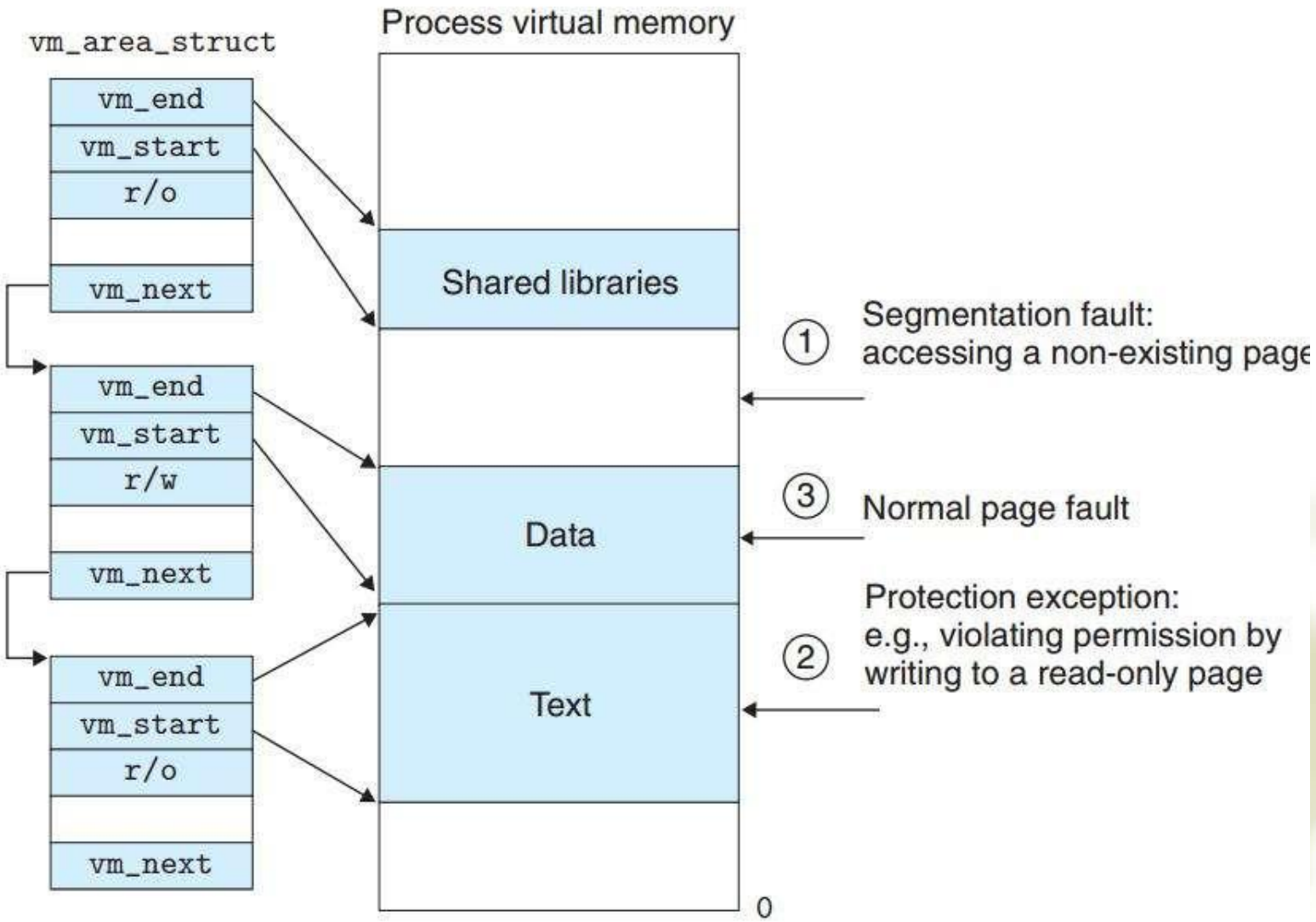
请页机制—实现虚存管理的重要手段

进程运行时，CPU访问的是用户空间的虚地址。

Linux仅把当前要使用的少量页面装入内存，需要时再通过请页机制将特定的页面调入内存。

当要访问的页不在内存时，产生一个页故障并报告故障原因。

请页机制—实现虚存管理的重要手段



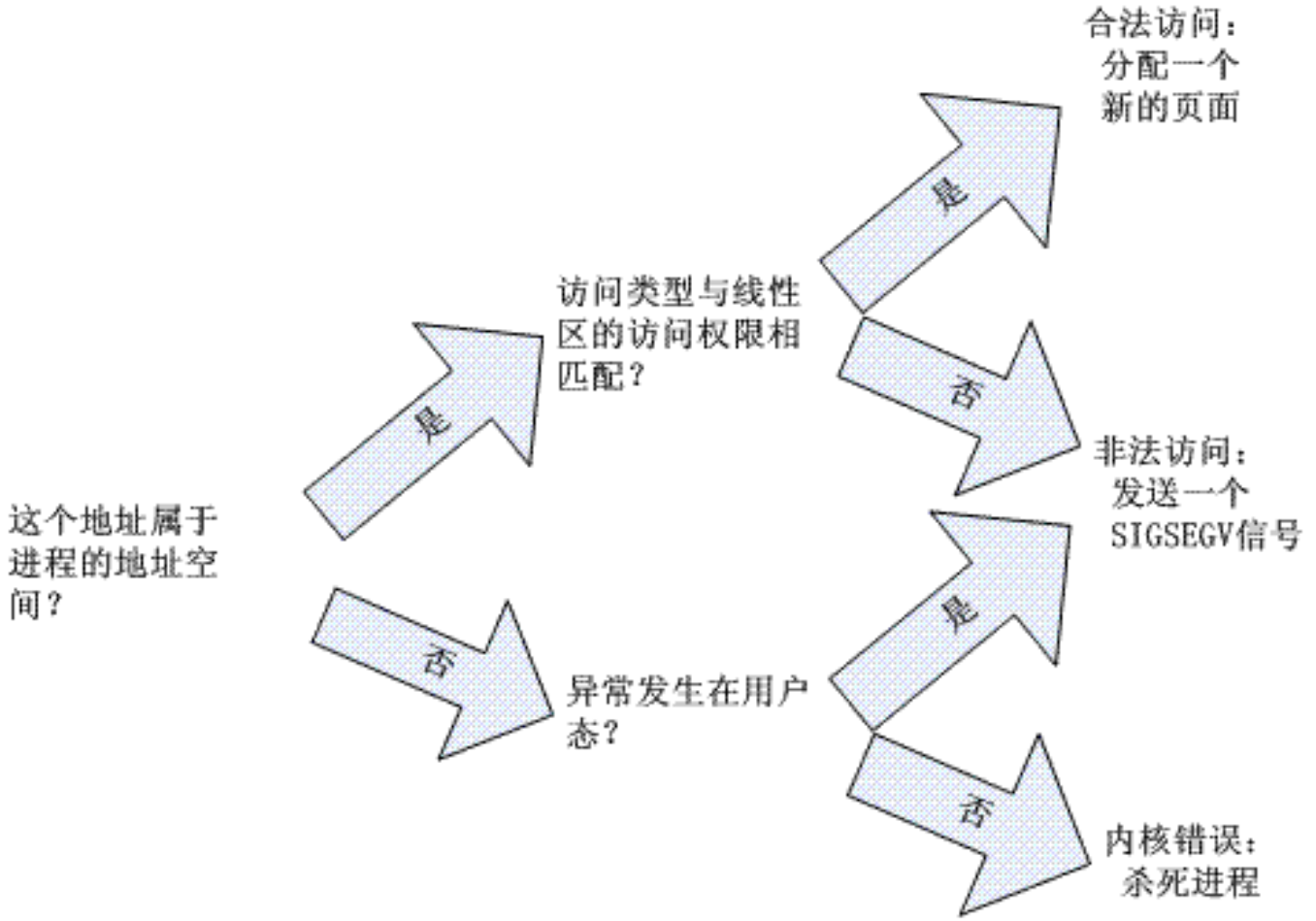
缺页异常处理

Linux的缺页（Page Fault）异常处理程序区分以下两种情况：是由编程错误所引起的异常还是缺页引起的异常；

如果是编程引起的异常，还发在内核态，毫不含糊的杀死该进程，如果发生在用户态，说明是无效的内存引用，程序也停止执行；

如果是缺页引起的异常，而且有合法的权限，则进入缺页异常处理程序。

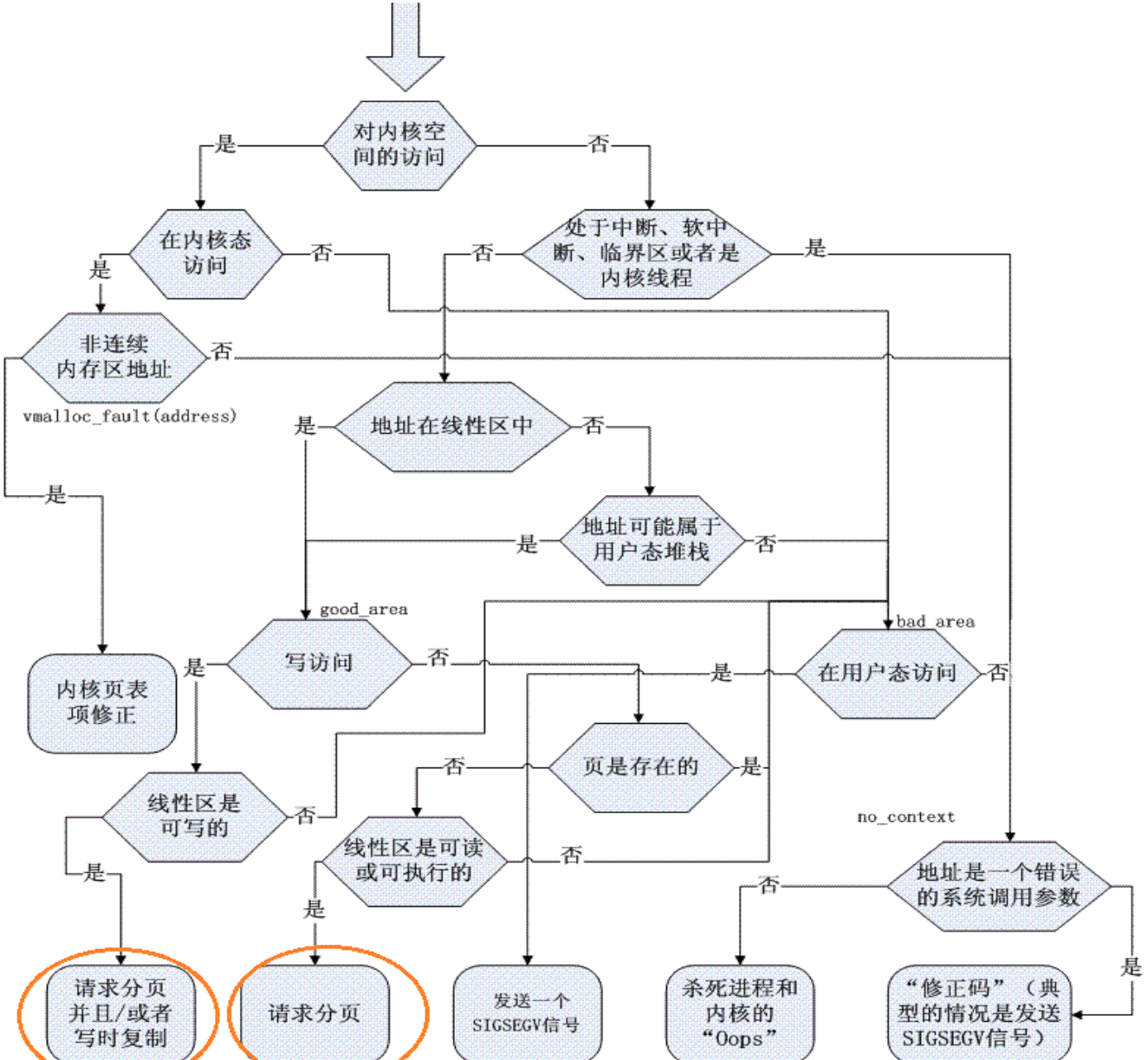
缺页异常处理


这个地址属于进程的地址空间？
是
否
访问类型与线性区的访问权限相匹配？
是
否
合法访问：分配一个新的页面
非法访问：发送一个SIGSEGV信号
异常发生在用户态？
是
否
内核错误：杀死进程

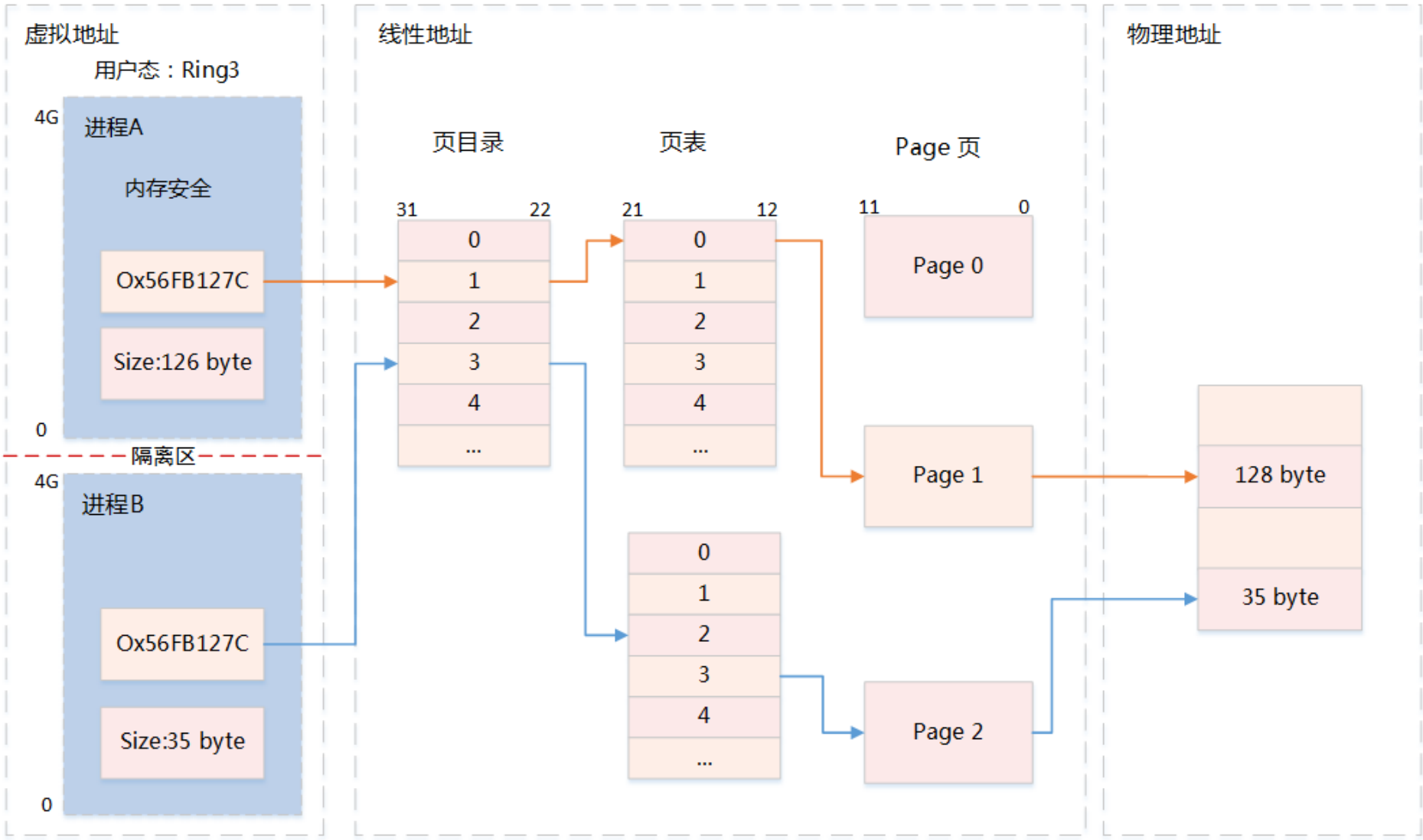
缺页异常处理流程

从图看出，很多分支都是处理缺页发生时异常情况，只有两个分支是真正的请求分页处理，大家可以先查看这两个正常情况处理的分支，然后，再看异常情况的处理。要真正深入了了解其处理过程，请阅读do_page_fault()函数。

缺页异常处理流程


对内核空间的访问
是
否
在内核态访问
是
否
非连续内存区地址
是
否
vmalloc_fault(address)
处于中断、软中断、临界区或者是内核线程
否
是
地址在线性区中
是
否
地址可能属于用户态堆栈
是
否
good_area
写访问
是
否
内核页表项修正
线性区是可写的
是
否
请求分页并且/或者写时复制
请求分页
bad_area
在用户态访问
是
否
页是存在的
是
否
线性区是可读或可执行的
是
否
发送一个SIGSEGV信号
no_context
地址是一个错误的系统调用参数
否
是
杀死进程和内核的“Oops”
“修正码”（典型的情况是发送SIGSEGV信号）

用户进程访问内存分析


虚拟地址
用户态：Ring3
4G
进程A
内存安全
Ox56FB127C
Size:126 byte
0
隔离区
4G
进程B
Ox56FB127C
Size:35 byte
0
线性地址
页目录
页表
Page 页
31
22
21
12
11
0
0
1
2
3
4
...
0
1
2
3
4
...
0
1
2
3
4
...
Page 0
Page 1
Page 2
物理地址
128 byte
35 byte

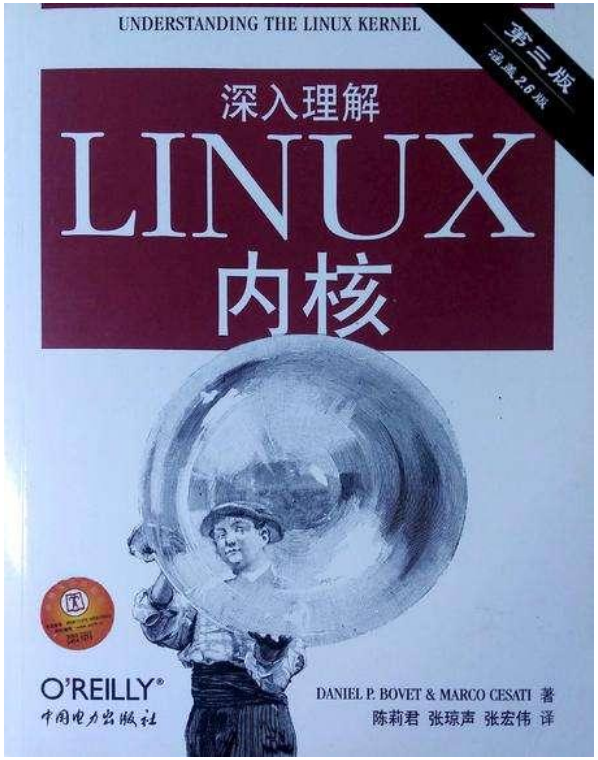
用户进程访问内存分析

1. 用户态进程独占虚拟地址空间，两个进程的虚拟地址可相同。

2. 在访问用户态虚拟地址空间时，如果没有映射物理地址，通过请页机制发出缺页异常。

3. 缺页异常陷入内核，分配物理地址空间，与用户态虚拟地址建立映射。如何分配物理内存空间，请关注下一讲。

参考资料



深入理解Linux内核 第三版第八、九章

带着思考离开

1. 什么是内存泄漏，为什么会发生内存泄漏
2. 什么是野指针，为什么会出现野指针？
3. 通过mmap（）进行 内存映射，多进程是否安全？

谢谢大家！

THANK YOU