内存管理

内存管理

读书

Publish Date: 2021-08-11

虚拟内存管理

进程地址空间

VMA(Virtual Memory Area)

一个VMA是一段连续的虚拟内存，我们可以通过字符设备/proc/pid/maps查看VMA的信息。

这里我们可以看到maps包括了每个VMA的起始地址和结束地址，还有权限。

addr	permisstion	offset	devices	inode	path

```
aaaac2b11000-aaaac2bd7000 r-xp 00000000 fd:00 3932787                    /usr/bin/zsh
aaaac2be7000-aaaac2be9000 r--p 000c6000 fd:00 3932787                    /usr/bin/zsh
aaaac2be9000-aaaac2bef000 rw-p 000c8000 fd:00 3932787                    /usr/bin/zsh
aaaac2bef000-aaaac2c03000 rw-p 00000000 00:00 0
aaaae3e23000-aaaae4112000 rw-p 00000000 00:00 0                          [heap]
ffffa329c000-ffffa351c000 r--s 00000000 fd:00 4201810                    /usr/share/zsh/functions/Completion/Unix.zwc
ffffa352b000-ffffa353a000 r-xp 00000000 fd:00 3936681                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/computil.so
ffffa353a000-ffffa3549000 ---p 0000f000 fd:00 3936681                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/computil.so
ffffa3549000-ffffa354a000 r--p 0000e000 fd:00 3936681                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/computil.so
ffffa354a000-ffffa354b000 rw-p 0000f000 fd:00 3936681                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/computil.so
ffffa354b000-ffffa3570000 r--s 00000000 fd:00 4201977                    /usr/share/zsh/functions/Completion/Zsh.zwc
ffffa357f000-ffffa3581000 r-xp 00000000 fd:00 3936701                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/regex.so
ffffa3581000-ffffa3590000 ---p 00002000 fd:00 3936701                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/regex.so
ffffa3590000-ffffa3591000 r--p 00001000 fd:00 3936701                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/regex.so
ffffa3591000-ffffa3592000 rw-p 00002000 fd:00 3936701                    /usr/lib/aarch64-linux-gnu/zsh/5.8/zsh/regex.so
ffffa3592000-ffffa35ab000 r--s 00000000 fd:00 4202221                    /usr/share/zsh/functions/Zle.zwc
```

image-20210811150518670

addr是开始和结束地址，permisstion是权限，rwx分别是可读可写可执行。

offset是距离map_start位置的偏移量，如果不是从文件映射的这个位置等于0

devices是主设备fd : 副设备fd，两个16进制数，如果不是从文件映射的等于0:0

inode是文件number

path是文件位置

除了Mapping段是每个文件对应一个VMA，剩下的段（数据段代码段....)一个段对应一块VMA

VMA数据结构

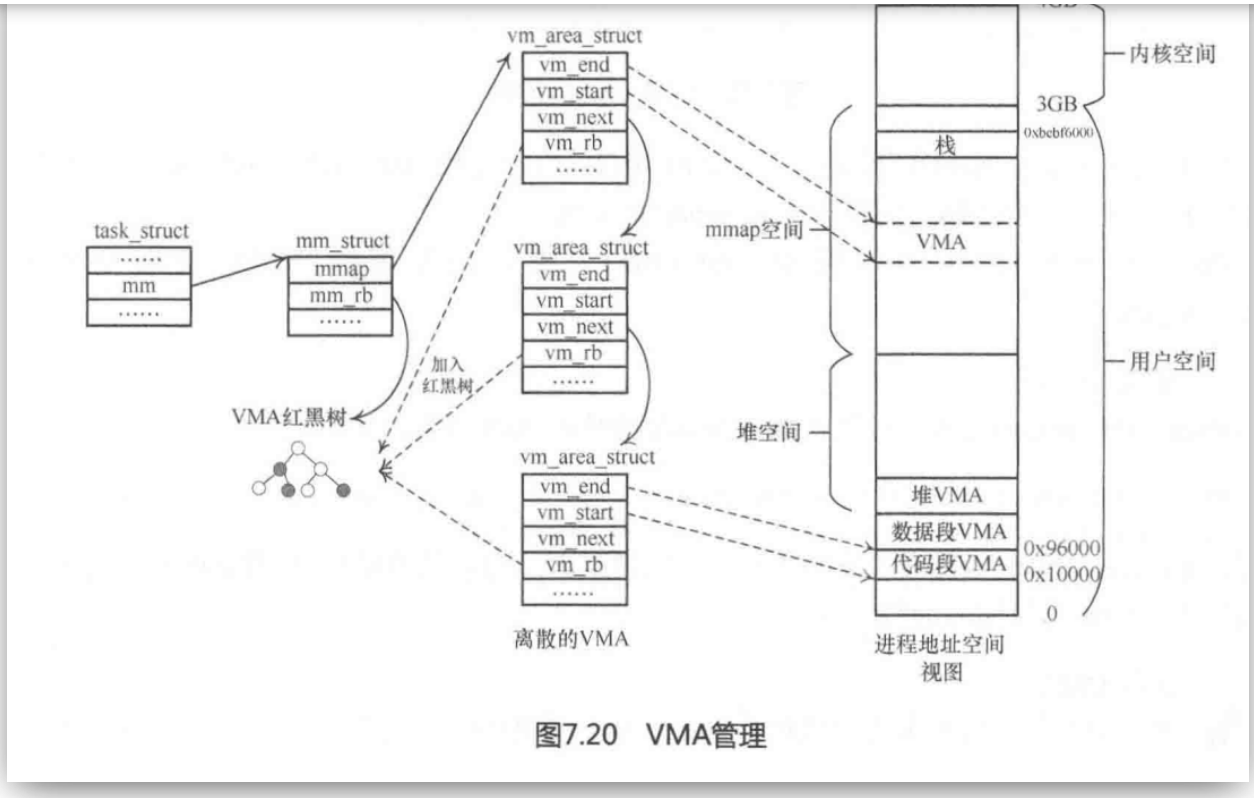

图7.20 VMA管理

image-20210812073029743

VMA同时插入了链表和红黑树中，用来提高查找效率。

查找

```c
struct vm_area_struct *find_vma(struct mm_struct* mm, unsigned long addr);
```

这个函数会沿着红黑树找到该虚拟地址所在的VMA

插入

```c
int insert_vm_struct(struct mm_struct *mm, struct vm_area_struct *vma)
```

插入时需要检查VMA是否与别的VMA可以合并

mmap

```c
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *addr, size_t length);
```

addr: 地址，如果让操作系统指定设置为NULL

length: 长度

prot: 权限PROT_EXEC PROT_READ PROT_WRITE PROT_NONE

flags: 标志位

MAP_SHARED MAP_PRIVATE决定内存区域是共享还是私有。共享时内存的修改会同步到文件中。私有时创建一个写时复制，常用来加载动态链接库，不会同步内存更改。

MAP_FIXED：要求按照给定的addr分配内存，否则失败。

MAP_POLULATE: 在文件映射时提前预读文件。

offset在文件映射时表示文件的偏移量。

常见用法

```c
mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0); //分配一页的内存

mmap(0, length, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE, fd, offset); //加载动态链接库

mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0); //父子进程共享内存

int fd = open("/tmp/shared_buffer", O_RDWR);
mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0); //不同进程共享内存
```

实现

在map内存区域插入新的VMA。

malloc brk sbrk

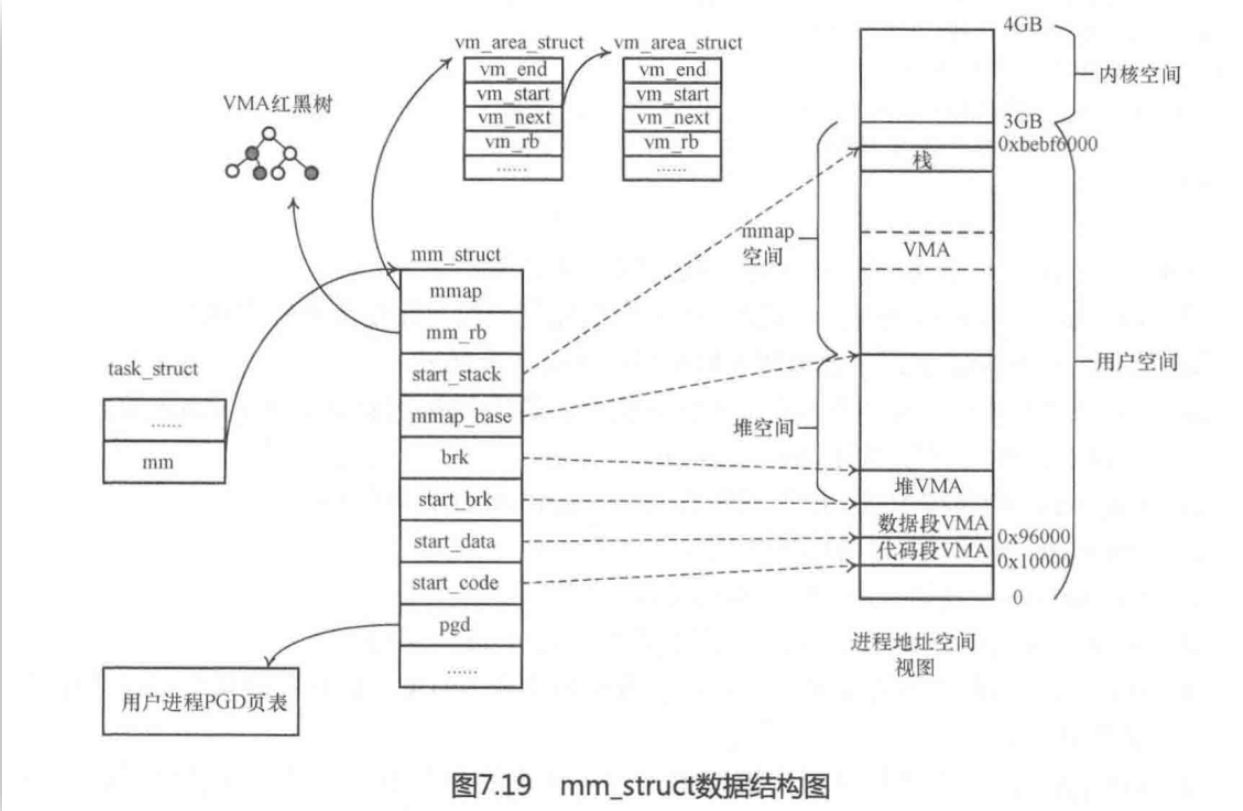

图7.19 mm_struct数据结构图

image-20210812073306877

从图中可以看到start_brk是数据段结束，就是堆开始的位置；brk是堆的结束位置

```c
int brk(void *addr);
void* sbrk(uintptr_t increment);
```

brk设置program break为地址addr

sbrk把program break增加increment(也可能是减少，取决于increment的正负)，然后返回之前的program break。可以用sbrk(0)查询当前的program break

为了避免重复的系统调用，malloc采用内存池的技术。当剩余的内存足够分配用户需要的大小时，将其分配给用户。否则调用mmap或者sbrk申请内存。

我们可以用一个简单的小实验

```c
char * addr1 = (char *)malloc(0);
char * addr2 = (char *)malloc(12);
char * addr3 = (char *)malloc(18);
printf("%ld --- %ld\n", addr2 - addr1, addr3 - addr2);

//结果是
//32 --- 32
```

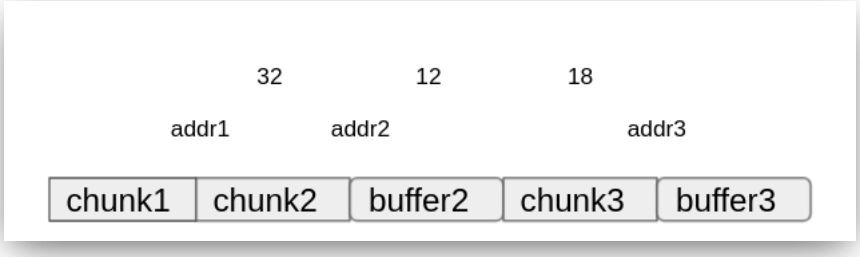

image-20210812075424503

Page Fault

ARM中有两个寄存器

`Fault Status Register, FSR` 存触发Page Fault的标志位

`Fault Address Register, FAR` 存触发Page Fault的虚拟地址

主要有几种情况

1. 匿名映射中断，发生在malloc和mmap，需要分配内存
2. 文件映射中断，发生在mmap读取文件，需要把文件从磁盘加载到内存
3. swap缺页中断，发生在页面被换出到磁盘上，把页面重新加载进内存
4. copy on write中断，发生在fork中，需要创建一个新的具有写权限的页给需要写操作的进程。

页面回收 LRU算法

内核中有五个LRU链表

- 不活跃匿名页面链表 LRU_INACTIVE_ANON
- 活跃匿名页面链表 LRU_ACTIVE_ANON
- 不活跃文件映射链表 LRU_INACTIVE_FILE
- 活跃文件映射链表 LRU_ACTIVE_FILE
- 不可回收页面链表 LRU_UNEVICTABLE

优先选择文件映射链表，因为文件映射不一定需要回写，除非需要对文件进行修改。而匿名映射链表是需要写入交换分区的。

Exercise5: 打印进程VMA

根据VMA的链表按顺序打印即可。我们知道vma的数据结构中有链表节点以及红黑树节点，我们任选一种都可以。为了简单，我们选择了链表节点并以此遍历。
此外，我们还需要通过 `pid_task` 找到PCB，再通过PCB找到mm，再找到vma。

```
#include <linux/init.h>
#include <linux/mm.h>
#include <linux/sched.h>

static int pid = 0; //默认打印自己
module_param(pid, int, S_IRUGO);

static void printit(struct task_struct *tsk)
{
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    int j = 0;

    unsigned long start, end, length;
    mm = tsk->mm;
    pr_info("mm = %p\n", mm);
    vma = mm->mmap;

    down_read(&mm->mmap_sem);
    pr_info
        ("vmas:\tvma\tstart\tend\tlength\n");

    while(vma){
        j++;
        start = vma->vm_start;
        end = vma->vm_end;
        length = end - start;
        pr_info("%d\t%p\t%lx\t%lx\t%ld\n",
                j, vma, start, end, length);
        vma = vma->vm_next;
    }
    up_read(&mm->mmap_sem);
}

static int __init my_init(void)
{
    struct task_struct *tsk;
    if(pid == 0){
        tsk = current;
        pid = current->pid;
    }else{
        tsk = pid_task(find_vpid(pid), PIDTYPE_PID);
    }
    if(!tsk){
        return -1;
    }

    pr_info(" Examining vma's for pid=%d, command=%s\n", pid, tsk->comm);
    printit(tsk);
    return 0;
}

static void __exit my_exit(void)
{
    pr_info("Module Unloading\n");
}

module_init(my_init);
module_exit(my_exit);
```

```
.my_vma.ko.cmd     .tmp_versions/     modules.order      my_vma.mod.c
.my_vma.mod.o.cmd  Makefile           my_vma.c           my_vma.mod.o
.my_vma.o.cmd      Module.symvers     my_vma.ko          my_vma.o
/mnt/lab5 # insmod my_vma.ko pid=0
my_vma: loading out-of-tree module taints kernel.
 Examining vma's for pid=721, command=insmod
mm = cc33e000
vmas:    vma     start   end     length
1        cc163058        10000   227000  2191360
2        cc1630b0        236000  239000  12288
3        cc163108        239000  25d000  147456
4        cc163000        bea98000        beab9000        135168
5        cc163160        bec48000        bec49000        4096
6        cc1631b8        bec49000        bec4a000        4096
7        cc163210        bec4a000        bec4b000        4096
print done
```

image-20210812133951477

Exercise6 实现mmap

非常抱歉延误了好几天。之前因为环境配置了太久所以没有做完。
实验的要求是在内核中分配一段内存，并可以映射到用户进程的地址空间。我们需要实现文件操作符的mmap接口，并在其中使用 [remap_pfn_range](https://www.kernel.org/doc/htmldocs/kernel-api/API-remap-pfn-range.html) 来映射这段物理内存。

这个函数的描述是"remap kernel memory to userspace"，也就是将内核空间的内存映射到用户空间，因此我们在自定义的mmap中使用这个函数就可以了，他的作用是新建vma节点。

首先在init和exit中分配和释放内存

```c
static int __init simple_char_init(void)
{
    // register a misc
    int ret = misc_register(&my_misc_device);
    if(ret){
        printk("failed to register misc device\n");
        return ret;
    }
    my_device = my_misc_device.this_device;
    printk("succeeded register char device: %s\n", DEV_NAME);

    // alloc a memory space
    buffer = kmalloc(4096, GFP_KERNEL);
    memset(buffer, 0, 4096);
    return 0;
}

static void __exit simple_char_exit(void)
{
    printk("removing device\n");
    misc_deregister(&my_misc_device);
    kfree(buffer);
}
```

然后实现mmap函数

```c
static int mydev_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if(remap_pfn_range(vma, vma->vm_start, virt_to_phys(buffer) >> PAGE_SHIFT, vma->vm_end -
vma->vm_start, vma->vm_page_prot))
        return -EAGAIN;
```

其他部分的代码

```c
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/slab.h>

#define DEV_NAME "my_dev"
#define MAX_DEVICE_BUFFER_SIZE 4096

static struct device *my_device;
char * buffer;


static int mydev_open(struct inode *inode, struct file *file)
{
    int major = MAJOR(inode->i_rdev);
    int minor = MINOR(inode->i_rdev);

    printk("%s: major=%d, minor=%d\n", __func__, major, minor);
    return 0;
}

static int mydev_release(struct inode *inode, struct file *file)
{
    return 0;
}

static ssize_t
mydev_read(struct file*file, char __user *buf, size_t lbuf, loff_t *ppos)
{
    int max_free = MAX_DEVICE_BUFFER_SIZE - *ppos;
    int need_read = max_free > lbuf ? lbuf : max_free;

    int ret = copy_to_user(buf, buffer + *ppos, need_read);
    int actual_read = need_read - ret;
    *ppos += actual_read;
    return actual_read;
}

static ssize_t
mydev_write(struct file* file, const char __user *buf, size_t count, loff_t *f_pos)
{
    int max_free = MAX_DEVICE_BUFFER_SIZE - *f_pos;
    int need_write = max_free > count ? count : max_free;

    int ret = copy_from_user(buffer + *f_pos, buf, need_write);

    int actual_write = need_write - ret;
    *f_pos += actual_write;

    return actual_write;
}


static const struct file_operations mydev_fops = {
    .owner = THIS_MODULE,
```

```
    .read = mydev_read,
    .write = mydev_write,
    .mmap = mydev_mmap,
};

static struct miscdevice my_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEV_NAME,
    .fops = &mydev_fops
};
```

测试程序代码

C

```c
#include <sys/mman.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>
#define DEV_NAME "/dev/my_dev"

int main()
{
    int fd = open(DEV_NAME, O_RDWR);
    if(fd < 0){
        fprintf(stderr, "open %s\n", DEV_NAME);
    }

    char *buffer = (char *)mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(buffer == MAP_FAILED){
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    sprintf(buffer, "hello world\n");
    printf("write hello world to dev\n");
    munmap(buffer, 4096);
    return 0;
}
```

然后我们打印设备看看写入是否成功了。观察到"hello world"即可。

Shell

```
echo /dev/my_dev
```

Exercise 7 映射用户内存

要求用get_user_pages映射用户内存，书上提示太少，然后在网上找了一个例子

内存管理