

**操作系统实验**

总结报告

|  |  |
| --- | --- |
| 姓名： | 张钧天 |
| 学号： | 2100013111 |
| 日期： | 2023.12.19 |

**目录**

[内容一：实验总结 1](#_Toc153826002)

[Lab 1 实验总结 1](#_Toc153826003)

[Lab 2 实验总结 1](#_Toc153826004)

[Lab 3 实验总结 2](#_Toc153826005)

[Lab 4 实验总结 2](#_Toc153826006)

[Lab 5 实验总结 2](#_Toc153826007)

[Lab 6 实验总结 3](#_Toc153826008)

[Lab 7 实验总结 3](#_Toc153826009)

[Lab 8 实验总结 3](#_Toc153826010)

[Lab 9 实验总结 4](#_Toc153826011)

[内容二：遇到的困难以及收获 4](#_Toc153826012)

[内容三：对课程或实验的意见和建议 4](#_Toc153826013)

[内容四：参考文献 4](#_Toc153826014)

## 内容一：实验总结

请对你本学期完成的实验进行总结，请分实验总结以下内容，包括但不限于：

1. 简要描述实验原理；

2. 实验结果展示；

3. 对实验任务的理解。

### Lab 1 实验总结

#### 实验原理

- 写完代码后，需要在Makefile里加入需要编译的文件名字

## sleep(easy)

系统调用sleep()，时间atoi()

## pingpong(easy)

父子进程pipe通信，将不使用的套接字关闭

```c

int pipe(int pipefd[2]);//0:read/1:write

write(p1[1], "!", 1); //发送给子进程

read(p2[0], buf, 1); //接收子进程

```

读取的时候，这个pipe的写端口有引用，并且读的时候为空，那么读就会阻塞–直到写释放才会读。

## primes (moderate)/(hard)

素数筛。

主进程将所有所要筛选范围内的数字(不包括1)按顺序发送给子进程，第一个子进程，收到的第一个数即为素数，剩下的数中筛除2的倍数在发送给下一个进程，后面过程以此类推，直至全部筛选完成。

可以尾端加-1表示结束

## find (moderate)

借鉴ls.c

```c

struct dirent de;

struct stat st;

fd=open(path,0);

fstat(fd,st);//get st.type(T\_FILE/T\_DIR)

p = buf + strlen(buf);

\*p++ = '/';

while (read(fd, &de, sizeof(de)) == sizeof(de))

de.inum!=0//个数

memmove(p, de.name, DIRSIZ);文件名de.name复制到p

```

## xargs (moderate)

fork之后exec(argv[0],argv);

分行读取命令行

#### 实验结果

100/100

#### 实验理解

完成五个函数的实用功能，帮助了解系统调用，一些实现方式也可参考已有函数。

### Lab 2 实验总结

#### 实验原理

## Using gdb (easy)

```shell

make qemu-gdb

gdb-multiarch

(gdb) source .gdbinit

b <func>

c //run until backtrace

n

p/x var

p/t $sstatus//register

backtrace

```

## System call tracing (moderate)

mask表示哪些函数需要跟踪

1. makefile添加trace.c到UPROGS（编译）

2. user/user.h 新增函数原型

3. user/usys.pl 新增系统调用支持

4. kernel/syscall.h 新增系统调用号码

5. kernel/syscall.c 新增系统调用映射名称数组

6. kernel/syscall.c 新增系统调用函数指针映射

7. kernel/syscall.c 新增extern内核调用函数

8. kernel/proc.h 新增结构体process中的mask字段

9. kernel/sysproc.c 新增sys\_trace()函数，保存参数至proc

```c

uint64

sys\_trace(void)

{

int mask;

argint(0, &mask);

struct proc \*p = myproc();

p->mask = mask;

return 0;

}

//`argint()`获取系统调用的参数mask，然后将其保存至当前进程的mask字段中

```

10. kernel/proc.c 修改fork()函数 np->mask = p->mask;

11. kernel/syscall.c 修改syscall()函数 mask满足时输出

12. user/trace.c

## Sysinfo (moderate)

```c

struct sysinfo {

uint64 freemem; // amount of free memory (bytes) （free\_mem\_num）

uint64 nproc; // number of process

};

```

1. makefile添加trace.c到UPROGS（编译）

2. user/user.h 新增函数原型

3. user/usys.pl 新增系统调用支持

4. kernel/syscall.h 新增系统调用号码

5. kernel/syscall.c 新增系统调用映射名称数组

6. kernel/syscall.c 新增系统调用函数指针映射

7. kernel/syscall.c 新增extern内核调用函数

8. kernel/proc.c 新增proc\_not\_unsed\_num函数，得到nproc

```c

int

proc\_not\_unsed\_num(void)

{

int nproc = 0;

for (struct proc \*p = proc; p < &proc[NPROC]; p++) {

if (p->state != UNUSED)

nproc++;

}

return nproc;

}

```

9. kernel/sysproc.c 新增sys\_sysinfo函数，导入sysinfo.h头文件

```c

uint64

sys\_sysinfo(void)

{

// user pointer to struct sysinfo

uint64 si\_addr;

argaddr(0, &si\_addr);

int nproc;

int freemem;

nproc = proc\_not\_unsed\_num();

freemem = free\_mem\_num();

struct sysinfo sysinfo;

sysinfo.freemem = freemem;

sysinfo.nproc = nproc;

struct proc \*p = myproc();

if (copyout(p->pagetable, si\_addr, (char \*)&sysinfo, sizeof(sysinfo)) < 0)//使用 `copyout` 将 `info` 传递到 user space

return -1;

return 0;

}

```

10. kernel/defs.h 新增free\_mem\_num()和proc\_not\_used\_num()函数

11. kernel/kalloc.c 新增free\_mem\_num()

```c

uint64

free\_mem\_num(void)

{

struct run \*r;

uint64 free\_num = 0;

acquire(&kmem.lock);

r = kmem.freelist;

while (r) {

free\_num++;

r = r->next;

}

release(&kmem.lock);

return free\_num \* 4096;

}

```

#### 实验结果

40/40

#### 实验理解

首先是使用gdb这一传统工具。

其次是实现系统调用，主要步骤有：

1. 修改makefile
2. user/user.h声明
3. user/usys.pl定义存根
4. kernel/syscall.h添加系统调用编号
5. 在kernel/syscall.c声明extern内核调用函数和编号映射表
6. 在 kernel/sysproc.c 中添加一个 sys\_xxxx（） 函数

最后，实验主要任务就是根据mask打印fork的追踪、获取系统的三个信息：

空闲内存由链表 kmem.list 给出，内存大小是宏定义的PGSIZE。

进程用proc表示，state是状态，ofile是fd打开的文件。

最后用copyput() 将内核数据传输到用户态。（系统调用通过寄存器传递参数，储存在用户态的上下文 trapframe 中）

### Lab 3 实验总结

#### 实验原理

## Speed up system calls (easy)

在创建进程时，就直接把进程的 pid 放入共享空间中，然后用户查询 pid 时，就不必通过 ecall 跳转到内核，省去了保存现场等开销。

1. 在proc 结构体中添加变量，存储共享页的物理地址：

2. 在 allocproc() 函数中，分配新的物理页并初始化：

```c

// Allocate a usyscall page.

if((p->usyscall = (struct usyscall \*)kalloc()) == 0){

freeproc(p);

release(&p->lock);

return 0;

}

memmove(p->usyscall, &p->pid, sizeof(int)); // 初始化为存储当前进程号

```

3. 在 kernel/proc.c 的 proc\_pagetable() 完成内存映射(mapping)：

```c

if(mappages(pagetable, USYSCALL, PGSIZE,

(uint64)(p->usyscall), PTE\_R | PTE\_U) < 0){

// 设定PTE\_U

// 如果内存映射失败，恢复以上页面：

uvmunmap(pagetable, TRAMPOLINE, 1, 0);

uvmunmap(pagetable, TRAPFRAME, 1, 0);

uvmfree(pagetable, 0);

return 0;

}

```

4. 在 freeproc() 函数中释放 p->usyscall 对应的物理页内存：

```c

if(p->usyscall)

kfree((void\*)p->usyscall);

p->usyscall = 0;

```

5. 在 proc\_freepagetable() 中释放页表中对应的页表项：

```c

uvmunmap(pagetable, USYSCALL, 1, 0);

```

## Print a page table (easy)

写一个通过 dfs 打印树的函数。

```c

// Recursively print page table information:

void vmprint(pagetable\_t pagetable, int level){

for(int i = 0; i < 512; i++){

pte\_t pte = pagetable[i];

if((pte & PTE\_V) && (pte & (PTE\_R|PTE\_W|PTE\_X)) == 0){

// this PTE points to a lower-level page table.

uint64 child = PTE2PA(pte);

for(int t=0;t<level;t++){ // 对应页表的级数level进行输出

printf(" ..");

}

printf("%d: pte %p pa %p\n", i, pte, child);

vmprint((pagetable\_t)child, level + 1); // 递归过程

}else if(pte & PTE\_V){

uint64 child = PTE2PA(pte);

for(int t=0;t<level;t++){

printf(" ..");

}

printf("%d: pte %p pa %p\n", i, pte, child);

}

}

}

```

## Detecting which pages have been accessed (hard)

实现一个 pgaccess() 函数，检测从上次调用这个函数开始，页表是否被访问过。

#define PTE\_A (1L << 6) //【设定访问位】

```c

int

sys\_pgaccess(void)

{

// lab pgtbl: your code here.

//先提取一下参数

struct proc\* p =myproc();

uint64 usrpge\_ptr;//待检测页表起始指针

int npage;//待检测页表个数

uint64 useraddr;//稍后写入用户内存

argaddr(0,&usrpge\_ptr);

argint(1,&npage);

argaddr(2,&useraddr);

if(npage>64)

{

return -1;

}

uint64 bitmap=0;

uint64 mask=1;

uint64 complement=PTE\_A;

complement=~complement;

int count=0;

​

for(uint64 page =usrpge\_ptr;page<usrpge\_ptr+npage\*PGSIZE;page+=PGSIZE)

{

pte\_t\* pte = walk(p->pagetable,page,0);

if(\*pte&PTE\_A)

{

bitmap=bitmap|(mask<<count);

\*pte=(\*pte)&complement;

}

count++;

//printf("bitmap:%p\n",bitmap);

​

}

copyout(p->pagetable,useraddr,(char\*)&bitmap,sizeof(bitmap));

​

return 0;

}

```

- 用argaddr()和argint()解析参数。

- 对于输出位掩码，在内核中存储一个临时缓冲区并在填充正确的位后将其复制给用户（通过copyout()）更容易。

- 如果PTE\_A已设置，在检查后务必清除它。否则，将无法确定自上次调用pgaccess()以来是否访问了页面（该位将被永久设置）

#### 实验结果

46/46

#### 实验理解

目的是理解页表的实现。

Trampoline 页面在地址空间最高的位置，随后是每个进程对应的内核栈，每个栈之间都有一个 Guard page ，该页的 PTE\_V 设置为 0，用于避免缓冲区溢出。

实验1利用“在特定的只读区域共享用户态和内核态数据”，消除系统调用的上下文切换，优化了getpid。

实验2实现通过dfs直观打印页表形态。

实验3利用walk检查页表项的access位。

### Lab 4 实验总结

#### 实验原理

## RISC-V assembly（easy）

```shell

make fs.img

```

## Backtrace (moderate)

1. 在kernel/defs.h新增void backtrace(void);

2. 在kernel/riscv.h新增获取frame pointer的方法

```c

static inline uint64

r\_fp()

{

uint64 x;

asm volatile("mv %0, s0" : "=r" (x));

return x;

}

扩张是 -16，而回收是 +16

栈帧中从高到低第一个 8 字节 `fp-8` 是 return address，也就是当前调用层应该返回到的地址。

栈帧中从高到低第二个 8 字节 `fp-16` 是 previous address，指向上一层栈帧的 fp 开始地址。

```

3. 在kernel/printf.c新增backtrace函数

```c

void backtrace() {

uint64 fp = r\_fp();

while(fp != PGROUNDUP(fp)) { // 如果已经到达栈底

uint64 ra = \*(uint64\*)(fp - 8); // return address

printf("%p\n", ra);

fp = \*(uint64\*)(fp - 16); // previous fp

}

}

通过宏 PGROUNDUP 和 PGROUNDDOWN 分别进行向上对齐和向下对齐

```

4. 在kernel/sysproc.c中的sys\_sleep和kernel/printf.c的panic调用函数

## Alarm(hard)

1. 在kernel/proc.h增加struct proc的相关变量

```c

struct proc {

...

int interval; // 从 sigalarm 开始，该进程能运行多少个 ticks

void (\*handler)(); // 指向处理函数的函数指针

int handler\_lock; // 自旋锁，防止对处理程序的重入调用（有点像关中断的动作）

int ticks; // 从 sigalarm 开始，该进程移进运行了多少个 ticks

struct trapframe \*trapframe\_copy; // 保存现场的一份 copy

};

```

2. 在kernel/proc.c添加alloc和free动作

```c

static struct proc\*

allocproc(void)

{

...

p->interval = 0;

p->handler = 0;

p->handler\_lock = 0;

p->ticks = 0;

if ((p->trapframe\_copy = (struct trapframe\*)kalloc()) == 0) {

freeproc(p);

release(&p->lock);

}

return p;

}

static void

freeproc(struct proc \*p)

{

...

if (p->trapframe\_copy) // 释放现场的copy的空间

kfree((void\*)p->trapframe\_copy);

}

```

3. 在kernel/trap.c保存现场、调用handler

```c

void

usertrap(void)

{

...

// give up the CPU if this is a timer interrupt.

if(which\_dev == 2) {

++(p->ticks);

if (p->interval != 0 && p->ticks == p->interval) {

if (p->handler\_lock == 0) {

p->handler\_lock = 1;

//copy\_trapframe(p->trapframe, p->trapframe\_copy);

\*p->trapframe\_copy = \*p->trapframe;

p->trapframe->epc = (uint64)p->handler;

}

}

yield();

}

}

```

4. 在user/user.h、user/usys.pl、kernel/syscall.h、kernel/syscall.c添加两个声明

```c

int sigalarm(int, void(\*)(void));

int sigreturn(void);

entry("sigalarm");

entry("sigreturn");

#define SYS\_sigalarm 22

#define SYS\_sigreturn 23

extern uint64 sys\_sigalarm(void);

extern uint64 sys\_sigreturn(void);

static uint64 (\*syscalls[])(void) = {

...

[SYS\_sigalarm] sys\_sigalarm,

[SYS\_sigreturn] sys\_sigreturn,

};

```

5. 在kernel/sysproc.c中添加系统调用

```c

uint64

sys\_sigalarm(void) {

int interval;

uint64 handler;

argint(0, &interval);

argaddr(1, &handler);

myproc()->interval = interval;

myproc()->handler = (void(\*)(void))handler;

return 0;

}

uint64

sys\_sigreturn(void) {

//copy\_trapframe(myproc()->trapframe\_copy, myproc()->trapframe);

//myproc->trapframe=myproc()->trapframe\_copy;

memmove(p->trapframe, p->trapframe\_copy, 512);

myproc()->ticks = 0;

myproc()->handler\_lock = 0;

return myproc()->trapframe->a0;

}

```

6. 在 `Makefile` 中添加 `alarmtest` 的指令声明

#### 实验结果

95/95

#### 实验理解

实验1回顾编译内容；

实验2添加 backtrace 功能，打印出调用栈，用于调试。需要注意栈的特性。

实验3实现进程的时间分片。具体实现sigalarm和sigreturn，并考虑时钟机制的各种注意事项。

### Lab 5 实验总结

#### 实验原理

## Implement copy-on write (hard)

```c

#define PTE\_COW (1L << 8) // copy on write page

```

- 分配页面，引用数初始化为1

- 一开始每一页没有调用kalloc的也置1

- kfree时-1，为0时释放页面

- uvmvopy无需复制物理页，只需复制父进程页表，去除写权限，标记为cow页，增加引用数

- 处理page fault（write）r\_scause() == 15：

1. 不合法的虚拟地址 return -1

2. 不是cow页面 return -1

3. 超过两个引用：引用数减1，复制新页面，修正权限

4. 一个引用：修正写权限

- copyout：当复制的目的地址(dstva)指向的页面是COW页面时调用以上相同函数处理

#### 实验结果

110/110

#### 实验理解

实现copy on write机制，注意一些细节即可。

修改引用计数（同时加锁应对多核）、修改pte权限、实现cowfault、处理copyout函数。

### Lab 6 实验总结

#### 实验原理

## Uthread: switching between threads（moderate）

- 首先定义一个`context`结构体保存线程上下文，并加入到`thread`结构体中。在上下文中只需要保存被调用者保存的寄存器，即`sp`和`s0-s11`，`ra`用来保存线程的返回地址

- 之后在`thread\_create`中加入初始化代码，使`ra`指向线程的入口函数，`sp`和`fp`指向栈底。注意栈底应该是`t->stack[STACK\_SIZE - 1]`，因为栈是从高地址向低地址增长的。

- 最后实现`thread\_switch`函数并在`thread\_schedule`中通过`thread\_switch((uint64)&t->context, (uint64)&next\_thread->context);`调用即可。

- `thread\_switch`需要对上下文进行保护和恢复，并通过设置`ra`寄存器和`ret`指令来恢复下一个线程的执行。

## Using threads（moderate）

pthread\_mutex\_t lock[NBUCKET]; // 定义锁

pthread\_mutex\_lock(&lock[i]); // 获取锁

pthread\_mutex\_unlock(&lock[i]); // 释放锁

for (int i = 0; i < NBUCKET; i++) pthread\_mutex\_init(&lock[i], NULL);// 初始化锁

## barrier（moderate）

```c

static void

barrier()

{

pthread\_mutex\_lock(&bstate.barrier\_mutex);

bstate.nthread++;

if(bstate.nthread == nthread){

bstate.round++;

bstate.nthread = 0;

pthread\_cond\_broadcast(&bstate.barrier\_cond);

}else{

pthread\_cond\_wait(&bstate.barrier\_cond, &bstate.barrier\_mutex);

}

pthread\_mutex\_unlock(&bstate.barrier\_mutex);

}

```

就是加锁然后判断到达屏障点的线程数，如果所有线程都到达了就调用`pthread\_cond\_broadcast`唤醒其他线程，否则就调用`pthread\_cond\_wait`进行等待。

#### 实验结果

60/60

#### 实验理解

初步了解多线程以及锁。

熟悉线程的添加、上下文切换、锁的初始化和使用、POSIX的条件变量。

### Lab 7 实验总结

#### 实验原理

## Memory allocator（moderate）

为每个CPU核心分配一个空闲链表，`kalloc`和`kfree`都在本核心的链表上进行，只有当当前核心的链表为空时才去访问其他核心的链表。通过这种策略就可以减少锁的争用，只有当某核心的链表为空时才会发生锁争用。

修改kinit、kfree和kalloc

## Buffer cache（hard）

- 创建一个大小为十三的哈希表，表中元素为cache池。每个池拥有一个锁，保证操作的原子性。

- cache块根据 blockno 哈希到不同池中。

- 读取新的磁盘块时，先到对应的cache池中寻找是否已缓存；若无，则先在自己的池中寻找空闲块；若无，则到其他池子寻找空闲块，找到后将该块转移到本池。

- 为cache块打上时间戳，每次使用时更新，寻找空闲块时，优先选择最久未被使用的空闲块。

- 块被释放时并不做另外的操作，留在原地。

#### 实验结果

80/80

#### 实验理解

一方面是改进空闲链表（内存池）与锁，另一方面是改进cache池与锁。

通过修改内存块策略，即可减少锁冲突。

后者相对实现难度更大，并且经常会因为一些设计的问题产生一个进程持有多把锁或者死锁的问题。

### Lab 8 实验总结

#### 实验原理

## Large files（moderate）

- 原始文件最大为12+256块。可以通过将一个直接块表中的项替换为一个二级块表来使系统支持大小为11+256+256\*256个块的文件。

- 修改宏与inode的定义

- 修改bmap，使其支持二级块表

```c

static uint

bmap(struct inode \*ip, uint bn)

{

...

bn -= NINDIRECT;

if(bn < NINDIRECT \* NINDIRECT){

// double indirect

int idx = bn / NINDIRECT;

int off = bn % NINDIRECT;

if((addr = ip->addrs[NDIRECT + 1]) == 0)

ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if((addr = a[idx]) == 0){

a[idx] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if((addr = a[off]) == 0){

a[off] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

return addr;

}

panic("bmap: out of range");

}

```

- 修改turunc释放二级块表对应的块

```c

static uint

bmap(struct inode \*ip, uint bn)

{

...

bn -= NINDIRECT;

if(bn < NINDIRECT \* NINDIRECT){

// double indirect

int idx = bn / NINDIRECT;

int off = bn % NINDIRECT;

if((addr = ip->addrs[NDIRECT + 1]) == 0)

ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if((addr = a[idx]) == 0){

a[idx] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if((addr = a[off]) == 0){

a[off] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

return addr;

}

panic("bmap: out of range");

}

```

## Symbolic links（moderate）

- symlink的系统调用就是创建一个inode，设置类型为T\_SYMLINK，然后向这个inode中写入目标文件的路径。

```c

static uint

bmap(struct inode \*ip, uint bn)

{

...

bn -= NINDIRECT;

if(bn < NINDIRECT \* NINDIRECT){

// double indirect

int idx = bn / NINDIRECT;

int off = bn % NINDIRECT;

if((addr = ip->addrs[NDIRECT + 1]) == 0)

ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if((addr = a[idx]) == 0){

a[idx] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if((addr = a[off]) == 0){

a[off] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

return addr;

}

panic("bmap: out of range");

}

```

- 在sys\_open中添加对符号链接的处理，当模式不是O\_NOFOLLOW的时候就对符号链接进行循环处理，直到找到真正的文件，如果循环超过了一定的次数（10），就说明可能发生了循环链接，就返回-1。

```c

uint64

sys\_open(void)

{

...

if(ip->type == T\_DEVICE && (ip->major < 0 || ip->major >= NDEV)){

...

}

if(ip->type == T\_SYMLINK){

if(!(omode & O\_NOFOLLOW)){

int cycle = 0;

char target[MAXPATH];

while(ip->type == T\_SYMLINK){

if(cycle == 10){

iunlockput(ip);

end\_op();

return -1; // max cycle

}

cycle++;

memset(target, 0, sizeof(target));

readi(ip, 0, (uint64)target, 0, MAXPATH);

iunlockput(ip);

if((ip = namei(target)) == 0){

end\_op();

return -1; // target not exist

}

ilock(ip);

}

}

}

if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){

...

}

```

- 要注意namei函数不会对ip上锁，需要使用ilock来上锁，而create则会上锁。

#### 实验结果

100/100

#### 实验理解

对文件系统修改，使其支持更大的文件以及符号链接。

对于大文件，主要是对二级块表的实现。

符号链接就是在文件中保存指向文件的路径名，在打开文件的时候根据保存的路径名再去查找实际文件。

### Lab 9 实验总结

#### 实验原理

## mmap（hard）

以mmap为例：

- 使用arg相关函数获得系统调用的6个参数；

- 在建立映射前，处理一些参数错误的状态；

- 在实验中我们只处理addr参数为0的映射，也就是内核来分配用户虚拟内存。需要在进程结构体中获得mmap字段(VMA数组)，从中找到一个空的VMA结构用于记录mmap的信息。

- 使用find\_freespace函数查找空闲的虚拟内存地址值

- 初始化VMA结构，并且将VMA结构中的struct file\*的ref字段增加(使用filedup函数)，防止文件提前释放。初始化时要注意perm字段的标识是perm<<1 | PTE\_U，添加PTE\_U是让用户进程能够访问，而perm<<1是PTE的权限位与mmap权限位之间的逻辑关系，就是往左移一位即可。

- 返回VMA的addr字段的值，这个值就是我们建立映射的虚拟内存的底部值，addr + len字段就是虚拟内存整个映射区域。

- 按照COW实验的做法，我们需要修改usertrap函数，在其中添加scause=13的判断，如下所示处理mmap缺页，首先是通过stval的地址(缺页异常的虚拟地址值)找到对应的VMA，并调用mmap\_fault函数处理mmap缺页。

munmap的要点：

- 通过munmap的addr参数，找到指定的VMA结构，通过判断参数len与VMA的len确定是否能够完全释放。

- 能够完全释放，这就比较简单，直接使用uvmunmap函数将所有页面释放，并清除物理页，将struct file\*字段的ref引用数减一，并清空VMA结构体。

- 不能够完全释放，判断addr是否从VMA映射的起始位置开始释放。如果addr = m->addr，就需要对应的修改len与addr字段；反之，将addr以后的映射区域全部取消。

#### 实验结果

140/140

#### 实验理解

实现mmap与munmap这两个系统调用，根据其机制进行细节处理。

## 内容二：遇到的困难以及收获

简要描述自己在实验中遇到的难题和解决方法，以及在实验中的收获与感想。

在添加系统调用时，需要考虑很多繁杂步骤，并认识为什么要那样做。我仔细对比了代码的改变，也查找了一些语句的具体功能，加深系统性理解。

实验4对我非常困难，尤其是alarm部分，我查找了许多博客资料才能搞明白如何实现那些具体的机制。

例如实验7，尚有一些机制没有理得特别清楚，解决的办法是回顾基础的一些理论知识，结合攻略来进行优化，实际在脑海中模拟运行整个过程。

在每次做lab时，我也有意识地用markdown语言写一些自己的理解，这对现在撰写实验报告也很有帮助。

一些博主也会在通过测试的基础上进行优化改进，对实验进行拓展，从中获得乐趣。

整个任务也有很多小提示，而且它对每个小的项目也明确区分了难度等级，结合xv6 book，对于整个操作系统的理解都收获良多，这是一个非常高质量的lab。

## 内容三：对课程或实验的意见和建议

欢迎同学们对本课程或实验提出宝贵意见和建议。

在已有介绍对现阶段一些具体经典操作系统的实操机制的同时，还可以适当增加一两节前沿专题课程。

对于实验课，可以增加一些要点的讲解。

## 内容四：参考文献

分别列出每个实验所参考的文章或者博客：

<https://th0ar.gitbooks.io/xv6-chinese/content/index.html>

Lab1:

<https://zhuanlan.zhihu.com/p/426506447>

Lab2:

<https://zhuanlan.zhihu.com/p/426507155>

<https://www.misaka-9982.com/2023/03/14/6-S081-Lab-2-System-Calls/>

Lab3:

<https://blog.csdn.net/weixin_53215555/article/details/127538392>

<https://ttzytt.com/2022/07/xv6_lab3_record/index.html>

<https://www.cnblogs.com/weijunji/p/14338430.html>

Lab4:

<https://blog.miigon.net/posts/s081-lab4-traps/>

<https://blog.csdn.net/Ryansior/article/details/127871556>

Lab5:

<https://xiaodongfan.com/6-S081-Lab5.html>

Lab6:

<https://www.cnblogs.com/weijunji/p/xv6-study-11.html>

<https://clownote.github.io/2021/04/27/xv6/Xv6-Lab-Multithreading/>

Lab7:

<https://zhuanlan.zhihu.com/p/426507542>

<https://blog.miigon.net/posts/s081-lab8-locks/>

Lab8:

<https://www.cnblogs.com/weijunji/p/xv6-study-14.html>

Lab9:

<https://zhuanlan.zhihu.com/p/655379016>

<https://www.cnblogs.com/weijunji/p/xv6-study-15.html>

注意：在实习报告完成后，请同学们在目录部分按右键更新域。