同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 10

2151767 薛树建

日期：2023 年 8 月 18 日

目录

Lab10：mmap

1. mmap(hard)

1) 实验目的

本实验要求我们在 xv 实现 mmap 与 munmap 的系统调用，mmap 的声明如下：

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

addr 为映射地址，在本实验中 addr 始终为 0，这表明将会有内核来决定映射文件的虚拟地址；length 是映射的内存长度；prot 是映射对应的权限；flags 是映射区域的标志，分为 shared 和 private ，如果是 shared ，那么最终对文件进行的修改会写回外存；fd 是文件描述符；off 是文件的偏移量。

munmap 的声明为：munmap(addr, length)，它应该删除指定地址范围内的 mmap 映射。

内存映射文件的好处

2) 实验步骤

a) 首先我们应当添加相应的系统调用，在 user.h，usys.pl，syscall.c 与 syscall.h 中添加对应字段，与前面的操作几乎都一样，此处不再赘述。

b) 我们在 proc.h 中定义一个结构体 VMA 用来 mmap 申请的虚拟内存空间。

```
#define VMA_MAX 16
struct VMA{
    int valid;          //有效位，当值为 0 时表示无效，即为 empty
    uint64 addr;        //记录起始地址
    int len;            //长度
    int prot;           //权限（read/write）
    int flags;          //区域类型（shared/private）
    int offset;           //偏移量
    struct file* f;     //映射的文件
    uint64 mapcnt;      //（延迟申请）已经映射的页数量
};
```

其参数与 mmap 的参数一致，多出的部分 valid 表示是否有效；f 则为映射的文件指针；mapcnt 为映射的页的数量

同时我们需要在 proc 结构体中去具体添加对应的 vma 数组。

```
char name[16];                  // Process name (
struct VMA vma[VMA_MAX];//虚拟内存空间
```

c) 之后我们需要在 allocproc 中对进程的虚拟内存空间进行初始化，

```
//初始化进程中的VMA
for( int i=0 ; i<VMA_MAX ; i++ )
 {
   p->vma[i].valid = 0;
   p->vma[i].mapcnt = 0;
 }
return p;
```

d) 接下来我们去实现 sys_mmap 函数
首先是获取对应参数

```
sys_mmap(void){
  uint64 addr;//映射地址
  int length;//长度
  int prot;//权限（read/write）
  int flags;//标志位
  int vfd;//对应的文件描述符
  struct file* f;//文件指针
  int offset;//偏移
  uint64 err = 0xffffffffffffffff;//错误时返回

  // 获取系统调用参数
  if(argaddr(0, &addr) < 0 || argint(1, &length) < 0 || argint(2, &prot) < 0 ||
    argint(3, &flags) < 0 || argfd(4, &vfd, &f) < 0 || argint(5, &offset) < 0)
    return err;
```

接着去除一些不需要考虑的情况，例如本实验中 addr 与 offset 始终为 0，同时包括 flags 标志位与区域类型之间的差异

```
//本实验中二者均为0
if(addr != 0 || offset != 0 || length < 0)
  return err;

// 文件不可写时不允许可写与共享
if(f->writable == 0 && (prot & PROT_WRITE) != 0 && flags == MAP_SHARED)
  return err;
```

接着我们从进程的 sz 位置开始为其分配虚拟内存。但没有实质的分配物理页（惰性分配）。

```
  return err;

// 遍历查找未使用的VMA结构体
for(int i = 0; i < VMA_MAX; ++i) {
  if(p->vma[i].valid== 0) {
    p->vma[i].valid = 1;
    p->vma[i].addr = p->sz;
    p->vma[i].len = length;
    p->vma[i].flags = flags;
    p->vma[i].prot = prot;
    p->vma[i].f = f;
    p->vma[i].offset = offset;

    // 增加文件的引用计数
    filedup(f);

    p->sz += length;
    return p->vma[i].addr;//返回地址
  }
}
```

e) 接下来我们需要在 usertrap 中发生 page fault 时为之分配实质的页面。

```
#ifdef LAB_MMAP
// 读取产生页面故障的虚拟地址，并判断是否位于有效区间
uint64 fault_va = r_stval();//缺页的地址
if(PGROUNDUP(p->trapframe->sp) - 1 < fault_va && fault_va < p->sz) {//判断其是否在有效范围内
  if(mmap_handler(r_stval(), r_scause()) != 0)
    p->killed = 1;
} else
  p->killed = 1;
#endif
```

从 r_stval 中获取缺页的地址，判断其是否位于有效范围内。之后便调用 mmap_handler 去为之分配内存。具体实现如下:

```
// 根据地址查找属于哪一个VMA
for(i = 0; i < VMA_MAX; ++i) {
  if(p->vma[i].valid && p->vma[i].addr <= va && va <= p->vma[i].addr + p->vma[i].len - 1) {
    break;
  }
}
if(i == VMA_MAX)//找不到返回-1
  return -1;
```

首先根据虚拟地址寻找其对应的 vma。

```
int pte_flags = PTE_U;//分配内存，建立记录pte权限的flags
if(p->vma[i].prot & PROT_READ)
  pte_flags |= PTE_R;
if(p->vma[i].prot & PROT_WRITE)
  pte_flags |= PTE_W;
if(p->vma[i].prot & PROT_EXEC)
  pte_flags |= PTE_X;//pte_x为可执行的标志位
```

建立对应权限的 flags，便于后面建立映射时作为参数传入

```
// 读导致的页面错误
if(cause == 13 && vf->readable == 0) return -1;
// 写导致的页面错误
if(cause == 15 && vf->writable == 0) return -1;
```

文件本身就是不可读或者不可写的情况直接返回

```
void* pa = kalloc();
if(pa == 0)
  return -1;//分配失败
memset(pa, 0, PGSIZE);//将该块页赋值为0

// 读取文件内容
ilock(vf->ip);

int offset = p->vma[i].offset + PGROUNDDOWN(va - p->vma[i].addr);//实际上始终为0
int readbytes = readi(vf->ip, 0, (uint64)pa, offset, PGSIZE);

if(readbytes == 0) {// 什么都没有读到
  iunlock(vf->ip);
  kfree(pa);
  return -1;
}
iunlock(vf->ip);


if(mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)pa, pte_flags) != 0) {// 添加页面映射
  kfree(pa);
  return -1;
}
```

接着下为该虚拟内存分配物理页，先读取文件内容到对应的物理内存地址，接着将该物理地址与虚拟地址做映射。

f) 实现 munmap：找到地址范围的 VMA 并取消映射指定的页面。

首先还是先获取对应参数

```
uint64 addr;
int length;
if(argaddr(0, &addr) < 0 || argint(1, &length) < 0)//获取
  return -1;
```

接着我们遍历寻找该虚拟地址对应的 vma，根据提示，该虚拟地址要不是出现在区域起始位置，不然就是结束位置。分别进行判断，并对区域做相应处理。

如果位于起始位置，则应该将 vma 中的 addr 后移 length，len 减去 length;如果是位于结束位置，应该将 vma 的 len 减去 length。

```
for(i = 0; i < VMA_MAX; ++i) {
  if(p->vma[i].valid && p->vma[i].len >= length) {
    //根据提示要么起始位置，要么结束位置
    //but you can assume that it will either unmap
    //at the start, or at the end, or the whole
    //region (but not punch a hole in the middle of a region)

    //起始位置
    if(p->vma[i].addr == addr) {
      p->vma[i].addr += length;
      p->vma[i].len -= length;
      break;
    }
    //结束位置
    if(addr + length == p->vma[i].addr + p->vma[i].len) {
      p->vma[i].len -= length;
      break;
    }
  }
```

如果该页面已被修改同时映射为 MAP_SHARED，则应该先调用 filewrite 写回

```
// 将MAP_SHARED页面写回文件系统
if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0) {
  filewrite(p->vma[i].f, addr, length);
}
```

接着调用 uvmunmap 移除映射同时设置参数 1 来释放物理内存，当映射全部移除后，则应该减少 file 对应的引用计数，并将其 valid 置为无效

```
  // 移除映射
  uvmunmap(p->pagetable, addr, length / PGSIZE, 1);
  // 当前VMA中全部映射都被取消
  if(p->vma[i].len == 0) {
    fileclose(p->vma[i].f);
    p->vma[i].valid= 0;
  }
```

g) 由于是惰性分配，在 uvmunmap 与 uvmcopy 中添加

```
  panic("uvmunmap: walk");
if((*pte & PTE_V) == 0)
  continue;
```

以此来避免陷入 panic。

h) 根据提示，对 exit 函数修改，保证其运行的时候能够将已映射区域取消映射。取消映射的逻辑与 munmap 类似，先判断是否需要写入，接着调用 fileclose 减少 file 的引用数，接着取消映射并置为无效

```
// 将进程的已映射区域取消映射
for(int i = 0; i < VMA_MAX; ++i) {
  if(p->vma[i].valid) {
    if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0) {
      filewrite(p->vma[i].f, p->vma[i].addr, p->vma[i].len);
    }
    fileclose(p->vma[i].f);
    uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].len / PGSIZE, 1);
    p->vma[i].valid = 0;
  }
}
```

i) 同时我们需要去修改 fork 函数，使子进程能够继承父进程的 vma

```
for(i=0;i<VMA_MAX;++i){
  if(p->vma[i].valid){
    memmove(&np->vma[i], &p->vma[i], sizeof(p->vma[i]));//复制内存
    filedup(p->vma[i].f);//增加文件的引用
  }
}
```

j) 进行测试，结果如下：

```
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
```

```
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

3) 实验中遇到的问题和解决方法

本实验要求我们去实现 xv6 中的 mmap 与 munmap，难度还是挺大的，不过好在实验指导的提示都给了相应的需要调用的函数与思路。

实际实验过程中主要还是通过阅读源码来解决相应的问题。

4) 实验心得

在完成 xv6 lab10 mmap 实验后，我对内存映射有了更深入的了解，并且掌握了如何在 xv6 操作系统中使用 mmap 函数来实现文件映射到内存的功能。

关于内存映射，用户在使用内存映射来对文件进行修改时要比直接调用 write 函数的性能好，内存映射可以减少用户态与内核态之间切换带来的开销。缺点大概是管理（分配与释放）较为复杂。

总结

总之，通过完成本实验，我巩固了操作系统的相关知识，还提升了对内存映射的理解。

本实验总体评分如下：