mmap 实验报告

专　业：软　件　工　程

指　导　老　师：王　冬　青

年　级：大　学　二　年　级

学　号：2153061

姓　名：谢　嘉　麒

1. mmap (hard)

1.1.实验要求

本实验中mmap可以用多种方式调用，但需要实现与内存映射文件相关的子集功能。假设addr始终为0，表示内核应该决定映射文件的虚拟地址。mmap返回该地址，如果失败则返回0xffffffffffffffff。length是要映射的字节数，它可能与文件长度不同。post指示内存是否应该可读，可写或可执行。flags是MAP_SHARED，表示对映射内存的修改应该写回到文件，或者是MAP_PRIVATE，表示它们不应该写回。fd是要映射的文件的打开文件描述符，可以假设偏移量为0；

如果多个进程映射了相同的MAP_SHARED文件，它们不必共享物理页面；

实验要求实现足够的mmap和munmap功能，使mmaptest正常工作。

1.2.实验步骤

1.2.1.准备工作

实验需要为xv6添加mmap并实现将文件映射到地址空间的功能，其形式如下：

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

根据实验经验，首先要在准备工作中为各个文件添加内容：

（1）在Makefile文件的UPROGS添加环境变量$U/_mmaptest\

（2）在kernel/syscall.h中添加#define SYS_mmap 22和#define SYS_munmap 23两个宏定义；

（3）在kernel/syscall.c中添加extern uint64 sys_mmap(void);和extern uint64 sys_munmap(void);并且在static uint64 (*syscalls[])(void)中添加[SYS_mmap] sys_mmap和[SYS_munmap] sys_munmap两个系统调用；

（4）在kernel/sysproc.c中添加void *sys_mmap和int sys_munmap两个函数；

（5）在user/user.h中添加这两个函数的函数声明；

（6）在user/usys.pl中添加这两个系统调用的入口entry("mmap");和entry("munmap");

1.2.2.定义数据结构

根据xv6实验手册，需要在kernel/proc.h文件中加入VMA数据结构：

（1）首先添加宏定义#define NVMA 16和#define VMA_START (MAXVA >> 1)；

（2）添加VMA数据结构：

```
struct vma {
    uint64 start;
    uint64 end;
    uint64 length;
    uint64 off;
    int perm;
    int flags;
    struct file *file;
    struct vma *next;
    struct spinlock lock;
};
```

（3）start表示虚拟内存区域的起始地址；

（4）end表示虚拟内存区域的结束地址；

（5）length表示虚拟内存区域的长度；

（6）off表示文件中的偏移量；

（7）perm表示权限，指定虚拟内存区域的访问权限；

（8）flags表示标志位；

（9）file是与VMA关联的文件；

（10）next是指向下一个虚拟内存区域的指针；

（11）lock是VMA的自旋锁，在多线程环境中保护VMA的并发访问的锁；

1.2.3.VMA的分配

设置vma_alloc函数来寻找空闲VMA并进行分配：

（1）定义struct vma vma_list[NVMA];来保存VMA数据；

（2）利用for(int i = 0; i < NVMA; i++)遍历vma_list中每一个元素；

（3）打开第i个VMA的自旋锁，检查其是否为空闲，是则返回，不是则释放自旋锁并继续寻找；

（4）循环结束后仍未找到则触发panic；

1.2.4.sys_mmap部分

实验并不要求真正完成内存映射，而是实现在发生缺页异常时进行映射的惰性机制，因此sys_mmap的具体实现思路如下：

（1）首先，从用户态获取参数 addr、length、prot、flags、fd和offset。

（2）然后，检查 addr 和 offset是否为 0，如果不为 0，则设为 0。该实现要求 addr总是为 0，表示内核决定虚拟地址的映射位置，而 offset也被设为 0，表示从文件的起始位置开始映射。

（3）检查文件描述符 fd 是否有效，即文件是否已经打开，并检查相应文件是否具有读写权限，以及 prot 和 flags 的有效性。根据 prot 和 flags 设置相应的页表标志 pte_flag。

（4）创建一个 VMA（虚拟内存区域）结构体，并设置该结构体的相关字段，包括权限 perm、长度 length、偏移 off、文件 file、标志 flags 等。接着，通过调用 filedup 函数，对文件结构体进行引用计数的增加，以确保在 VMA 生命周期内文件描述符不会被关闭。

（5）在设置 VMA 的 start 和 end 字段时，先检查进程是否已经有 VMA，如果没有，则将 start 设为 VMA_START（一个常量值），并将 end设为 length + VMA_START。如果进程已经有 VMA，则将新的 VMA 放在已有 VMA 之后，即将 start设为前一个 VMA 结束位置的对齐地址（PGROUNDUP），end 设为 start + length。

（6）最后，函数返回分配的虚拟地址 addr，表示成功将文件映射到了虚拟内存中。

1.2.5.在usertrap中处理缺页异常

为了在usertrap中处理缺页异常，因此需要实现在给定虚拟地址va下分配内存并将文件数据加载到该内存区域，具体实现思路如下：

（1）首先获取当前进程的VMA链表的头指针，然后遍历VMA链表查找是否有一个VMA的地址范围包含了给定的虚拟地址va。找到则跳出循环。

（2）找到，则检查scause，判断是否满足该VMA的权限要求。若scause为13或15，则其没有读权限或写权限，返回-1。

（3）函数分配一个页大小的内存，使用kalloc()函数在内核中分配物理页，并初始化为0；

（4）调用mappages函数将分配的物理页映射到进程的页表中，实现虚拟地址va到物理地址的映射，并根据VMA的权限将相应的页表项标记为可读或可写。

（5）获取VMA对应的文件结构体指针f，对文件加锁，用readi函数从文件中读取数据到刚刚映射的内存区域，起始偏移为v->off + va - v->start，长度为一个页大小。

（6）释放文件锁，返回0。

将以上内容封装在函数mmap_alloc中，在usertrap函数中，若r_scause()为13或15，则插入语句：

```
else if((r_scause() == 13) || (r_scause() == 15)){
    if (mmap_alloc(r_stval(), r_scause()) != 0)
    {
        printf("mmap: page fault\n");
        p->killed = 1;
    }
}
```

1.2.6.sys_munmap实现

这一部分需要实现的内容较为复杂，其中将修改过的内存数据写回较为突出，其具体实现思路如下：

（1）首先检查是否需要写回，若VMA没有写权限或者有一个私有映射，则无需写回；

（2）若需要写回，则先检查给定的虚拟地址addr是否页对齐，即是否是页的起始地址；

（3）获取VMA对应的文件结构体指针f，由于写回操作采用磁盘块方式，每个磁盘块大小为BSIZE，而一个操作不超过MAXOPBLOCKS，因此max = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE；

（4）使用写回总字节数从i到n的循环进行写回操作，每次循环中计算当前写回字节数k，若超过max则设置为max；

（5）每次循环中用writei函数将内存数据写回道文件，并根据写入字节数更新i，最后释放文件锁和操作锁；

实现以上内容后，将其作为功能函数进行封装，以便之后再sys_munmap中使用。

之后则需要实现对内存映射区域进行解除映射的操作，其实现思路如下：

（1）函数首先从用户空间获取虚拟地址addr和长度length，获取失败返回-1；

（2）获取当前进程的VMA列表头结点v，遍历VMA列表，寻找与给定的虚拟地址addr对应的VMA区域，找到则停止遍历；没找到则说明虚拟地址没有被映射，函数返回-1；

（3）找到对应的VMA区域，继续检查是否在此区域起始地址或结束地址解除映射，不是则产生panic，因为解除映射操作必须在VMA区域的边界处进行；

（4）若解除映射操作在起始地址处进行，则首先进行写回，将修改过的内存数据写回到文件，然后通过uvmunmap函数将VMA区域从进程的页表中解除映射，根据解除映射的长度length更新VMA区域的信息；

（5）若解除映射在VMA区域结束地址处进行，只更新VMA区域长度即可；

（6）若解除映射长度等于VMA区域总长度，说明整个区域都需要解除映射，则释放文件资源，并从进程的VMA列表中删除该VMA。若解除映射长度小于VMA区域总长度，则更新VMA区域的起始地址，偏移量和长度；

（7）函数成功完成解除映射操作后返回0；

1.2.7.细节处理

完成以上步骤后运行并测试，会发现出现panic如下，且fork出现failed：

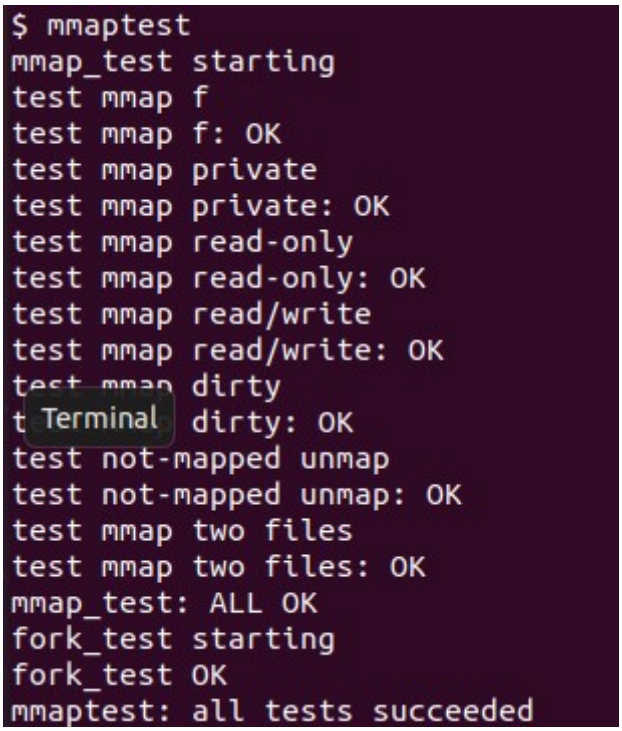
```
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
mmap: page fault
fork_test failed
panic: uvmunmap: not mapped
```

因此在vm.c文件中修改uvmunmap和freewalk两个函数，将其panic注释。

同时需要处理fork和exit函数。fork时需要将父进程的VMA复制给子进程，且进程退出时需要写回所有内容。

1.2.8.编译与检测

完成以上内容后，进入xv6目录下终端并进行编译，完成后输入mmaptest进行检测，结果如下：



1.3.实验中遇到的问题和解决方法

1.3.1.实现mmap后考虑到进程fork和exit

本实验中实现mmap后笔者最初并未考虑到fork和exit，因此测试中fork项出现失败。此疏忽警示笔者在实现和进程相关的内容时，都需要周全考虑到操作系统中所有与进程有关的操作，同时也需要综合前几个实验的内容来进一步理解并完成细节问题，防止出现错误。

1.4.实验心得

本次实验是xv6较为综合性的总结实验，用到了之前实验所了解和实现的大部分机制，以此为基础实现进一步的复杂功能。通过整个xv6实验，笔者认识到用户主要从user/user.h来对xv6进行使用，总体调用系统调用接口和C函数库。而内核则封装更为安全和复杂，其中包括进程，映射，内存页，中断等模块都在实验中多次遇到并进行实现。

通过实验，笔者对xv6系统有了整体而全面的认识，更是对操作系统的具体实现有了完成而具体的理解。具体的代码编写和模块的实现使笔者的编程能力得到了极大的提高，对实验手册和指导资料的阅读也令笔者对操作系统进行了系统的复习和知识内容的更新。