实验九

Mmap

目的：提供操作系统对地址空间的进一步控制，如进程之间共享内存，将文件映射进进程地址空间，以及成为用户页面错误设计方案的一部分（垃圾回收功能等）。

本实验要求实现文件内存映射的一部分功能。

**具体操作：**

void \*mmap(void \*addr, size\_t length, int prot, int flags,

int fd, off\_t offset);

munmap(addr, length)

步骤：

1. 准备工作：添加mmap与munmap系统调用，添加标志位宏定义。
2. Lazy分配：mmap不分配物理内存也不读取文件，而是在页面错误中进行。
3. 跟踪mmap在每一个进程的状态：定义类似VMA（虚拟内存区域）的数据结构，包括地址，长度、权限、文件等。注意到xv6没有实现内核内存分配，因此可以声明一个固定长度的VMA数组，需要空间就从其中分配。
4. 实现mmap：找到用户地址空间中没有分配的区域，将其分配给文件，并且创建一个VMA以包含映射空间的相关信息，放置在进程之中。VMA需要包含指向映射文件的指针，还要给文件的ref属性加一以保证文件被关闭后文件结构体不会消失（详情见filedup）
5. 解决page fault：分配一块物理内存，从文件中读入4096B（一块）的相关文件放置到该页中，并且做页表映射（记得设置正确的权限）。注意，使用readi读取文件（注意lock unlock），而且这会需要一个偏移offset参数。
6. 实现munmap：unmap VMA中对应的数据，还要卸除对应页表（uvunmap）。这个时候，如果所有映射的页都unmap了，还要减一文件ref属性。如果映射的页被写入了以及标记了MAP\_SHARED，就要写回到文件中（filewrite或许有帮助）
7. 理想情况下，只需要写回修改过的标记有MAP\_SHARED的页面。PTE中的dirty bit(D)表示对应块是否有写入。
8. 修改exit：如果munmap被调用，那么exit需要负责卸载用户的映射区域。
9. 修改fork：确保子进程拥有相同的VMA，还要增一ref，但是不要分配页表。

系统调用放在sysproc，相关操作放在vm。

Usertrap对应的解决方案：

* 首先，如何实现lazy？分配与释放的时候都按照一个块分配。释放的时候要注意，是按照页表进行地址的查找，找到了addr再释放，并设置这段空间已被释放。
* 第二，如何知道所有的map都已经被unmap？将每个VMA以一个PGSIZE为单位来存放，而非一整个size，这样避免了切分问题。但是要注意，这样会产生额外参数offset，在读取文件的时候，以及写入文件的时候要注意。还有，最终由于无法调节的锁bug原因，转而调用filewrite实现（用f->offset）控制offset。
* 第三，不是所有的map都被允许，如果传入的flag和文件的标识符不是被包含的关系，那么就不能够正常map。但是，如果是MAP\_PRIVATE模式，写就无所谓了。
* 第四：小心，mmaptest测试样例只有1.5个PAGE的’a’，但是却要读出两个PAGE的数据，并且test默认最后0.5PAGE为0，如果我们直接从file中读，是读不出的，只能够将内存超过部分置零。
* 第五（后来发现实际上要先考虑fork再考虑exit）：修改exit实际上就是munmap，那么把munmap单独放出来包装一层作为独立的函数就好啦。修改fork相反，就是mmap一下。还有一些些要注意，exit不会释放对应进程的资源，wait会，释放资源的时候，因为lazy，在unmap的时候会发现一些没有映射的页表，这个时候和下面的uvmcopy一样，略过就好。
* 第六：fork需要解决更多的问题，迎面而来的问题就是页表复制问题。页表中含有需要我们自身进行管理的vma部分。这里，我们采取的办法就是，卸下vma相关的页表（要写回file），但是不卸载vma（也不closefile），之后对新的进程进行vma的mmap（要进行filedup）。卸载之后，页表copy的时候会因为卸载和lazy策略的原因，会出现部分虚拟地址未映射的情况，因此需要uvmcopy略过这种情况。





