Lab 实验报告 X

Lab: Mmap

丁睿

dromniscience@gmail.com

更新：2020 年 12 月 22 日

目录

1 任务完成清单

Subtask	Done?	Time
Memory mapping	Y	6h

Grade: 140/140

我的作业已开放在这个网页上。我为 RISC-V xv6 的十次作业建立了一个统一主页，请访问此网页或者`https://github.com/dromniscience/MIT_xv6_lab_2020`。

2 详细情况 & 困难和收获

本次 lab 的设计方法相当多样。就任务本身而言，它同时整合了`Lab: System calls`、`Lab : Lazy Allocation` 和`Lab: File System`的内容。作为出发点，我首先实现了一个立即映射的草稿版本，并于测试通过后进一步实现了延迟分配策略。

我为这两个不同的版本用`git`各保存了一次历史。它们均能通过所有的测试样例——这反映了 mmap 的测试非常弱，至少没有测试映射一段甚至比 RAM 还大的区域的情况。以下我根据我完成此次 lab 的步骤，选择性地介绍其中重要的和某些细节性的内容。

2.1 Starters

这一步里我们只考虑实现立刻映射的`mmap`和`munmap`。这样不会发生 page fault。为了记录每个映射的文件，我们必须记录 Virtual Memory Area。一旦成功地调用，`mmap`立刻分配一块合适的区域，创建 VMA。VMA 中必要的字段如下:

```
/* kernel/proc.h */
struct vma {
  uint64 addr; // 0xffffffffffffffff: the slot is not used
  uint64 len;
  int prot;
  int shared;
  uint64 offset; // must not use offset in struct file
  struct file *pf;
};
```

`addr`标记该段起始的用户虚拟地址。它为`(uint64)-1`则意味着该`vma`未分配，如此我们节省了一个标记位。`len`表示该 VMA 段的大小。后续会提到一个 padding 的技巧，所以`len`并不一定等于实际的虚拟内存段大小。但是，通过优雅的向上舍入和向下舍入的运算，我们能够保证`unmap`时不会产生遗漏的页。`prot`标记读写权限。为方便计，直接存储`PROT_READ`和`PROT_WRITE`的组合字。`shared`标记当前段是否共享，非零的值 (`MAP_SHARED`) 表示共享，否则为私有。`offset`标记该段起始对应的文件中偏移。由于我们的读写不应当与共享同一打开文件表项的其他逻辑流同

步，因此必须单独维护该数据，并且不应当调用`fileread`和`filewrite`进行文件读写，因为它们使用`struct file`中的偏移量 (`off`)。最后，应当设置指向打开文件表条目的指针。

为简单计，每当`mmap`要构造一个 VMA，我们总是通过扩堆来创建。这个做法使得 PCB 中 VMA 数组的使用变得简单：只要找到一个空插槽，就可以从`myproc()->sz`开始分配。代价则是即便随后它被`munmap`了，在虚拟地址空间留下了一个巨大的“洞”，这个“洞”也不会被使用。但是注意到我们实际关心的是物理 RAM 的利用率，而这样的操作只会造成留下一些在进程结束前不再被使用的页表[1]。这个代价是可接受的。

在放置虚拟内存段时，我们采用了一个小 trick。注意由于可以交错地在首尾调用`munmap`，因此无论如何不能保证该段始终按页对齐 (page-aligned)，或者要求`off`是`PGSIZE`的倍数。那么我们应该怎么释放多余页呢？下面几幅图片说明了这一点。

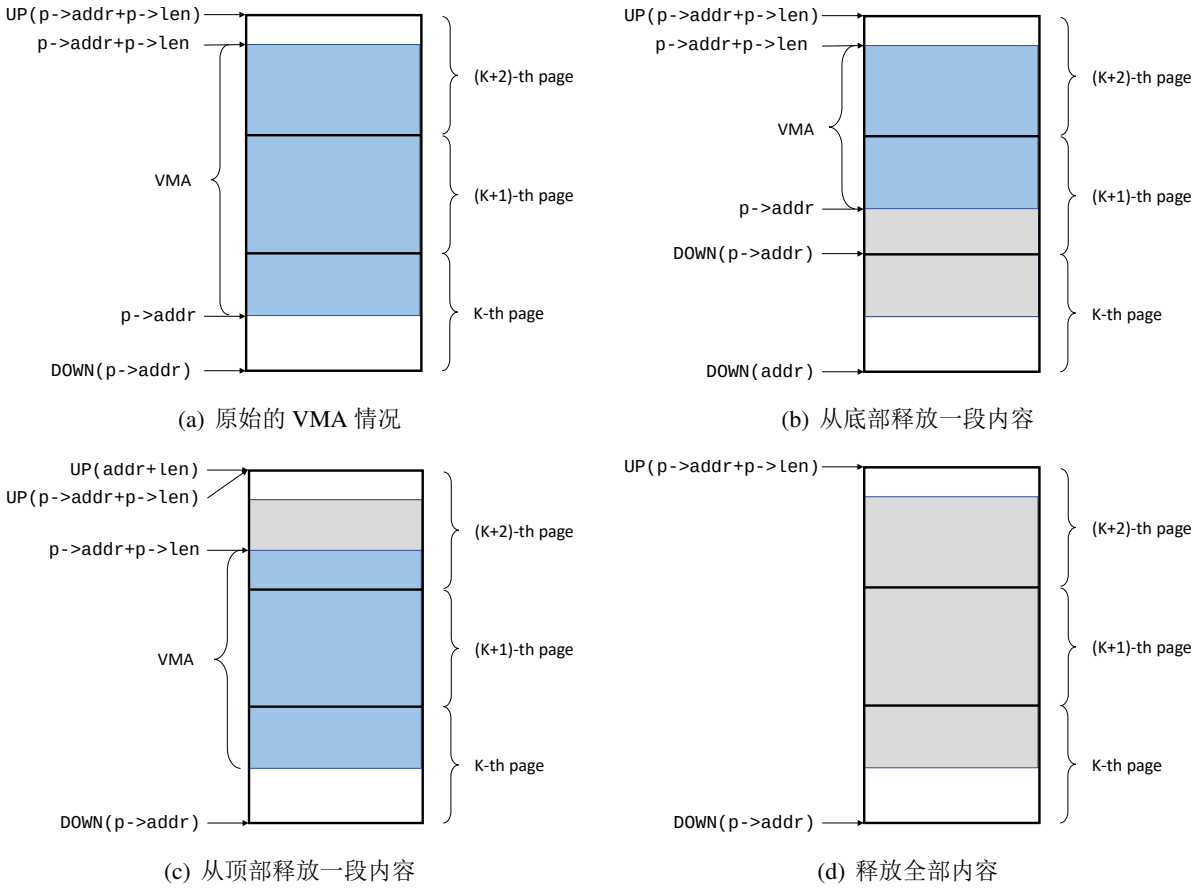

图 1: VMA 不同的释放情况。请注意，padding 的存在使得空白部分没有存放其他的进程数据。

如果很不幸地 VMA 所在页中还有其他的进程段 (比如进程在这里申请了堆)，那么`munmap`将陷入窘境——`munmap`完全不知道当前页是否能释放，因为如果没有对齐的话该页空余的部分可能存放了普通数据。为此，我们总是将 VMA 创建在若干完整且连续的虚拟页中。

如果遵循这个约定，那么对于起始地址是`p->addr`，终止地址是`p->addr+p->len`的段，它实际独占的页号总是`[PGROUNDDOWN(p->addr), PGROUNDUP(p->addr+p->len)-1]`。因此，针对上

[1] 回想一下 xv6 的页表页一旦创建，直到进程`exit`时调用`proc_freepagetable`才被释放。

图所示的三种释放情形，应当如此操作:

No.	Range of pages needed to be unmapped	Others
(b)	[PGROUNDDOWN(va), PGROUNDDOWN(p->addr))	update p->offset
(c)	[PGROUNDUP(p->addr + p->len), PGROUNDUP(va + len))	——
(d)	[PGROUNDDOWN(va), PGROUNDUP(va + len))	free VMA & close file

表 1: 据此很容易写出调用munmap函数的参数列表。va和len表示 VMA 的旧值。每种情况下更新p->addr和p->len的操作不再列出。

一旦打开文件，须立刻使用filedup增加文件的引用计数以防其失效。在fork出子进程时，同样应该增加引用计数 (当然别忘了复制 VMA 的内容)。最后在exit这个统一的进程退出出口中，注意释放所有的段并关闭文件。

你可能会考虑 VMA 段的管理是否还有疏漏之处。实际上，exec是一个危险的操作——如果一个进程尚未munmap掉自己映射的全部文件，exec就会产生一个“消失”的引用计数。可是，我们将此处的正确性交给程序员保证，尽管在exec中实现这样的检查和释放操作并不难。

一旦引入mmap和munmap，进程的有效虚拟地址空间就不再连续。因此，我们必须修改uvmcopy和uvmunmap以适应不连续的地址的情况。根据 Lab: Lazy Allocation 的经验，walkaddr也应该修改。但是请注意，它被用于查找sbrk分配的读写地址，而它们的数据段是立刻分配且连续的 (不考虑跨段的读写[2])，因此不需修改。

最后提到代码复用的问题。由于多处存在文件关闭和打开操作，可为它们单独封装一段代码，例如我的mapfileread和unmapfilewrite(为了和已有的文件读写函数区别)。同样，对于 VMA 的更新也可单独封装一个功能函数，方便维护。struct proc中 VMA 数组设置为静态长度MAXVMA(16)。

2.2 Lazy Allocation

第一阶段的实现的 memory mapping 机制已能得到满分。我们继续考虑对mmap采取延迟分配。为此，创建一个新的函数lazy_growproc。它仅在mmap中被调用，且参数须为正数。以下着眼分析缺页处理程序。

首先应该对读写权限进行检查，然后使用mappages在页表项中注册合适的权限[3]。在发起文件读时，应注意读取的范围。我们使用如下代码说明这一点：

```
1 /* kernel/trap.c */
2 mapfileread(p->vma + i, va, p->vma[i].offset + va - p->vma[i].
     addr, PGROUNDDOWN(va + PGSIZE) - va);
```

[2]真实的系统中往往存在跨段的读写。但此处由于mmap参数的第一个参数总是为零，我们就简单地排除这种可能——一个不希望打断堆段连续性的程序员当然要给一个mmap建议一个高地址。

[3]测试样例中甚至没有单独测试读写权限的正确性。一个可能原因是原来 xv6 的所有用户页默认都是可读可写可执行的。

```
/* kernel/sysfile.c */
void
mapfileread(struct vma *pvma, uint64 va, uint64 offset, uint64
    len)
{
    struct file *pf = pvma->pf;
    begin_op();
    ilock(pf->ip);
    readi(pf->ip, 1, va, offset, len);
    iunlock(pf->ip);
    end_op();
}
```

它遵守了一般磁盘读写的惯例：`begin_op`在 log 中预留`MAXOPBLOCKS`个条目，并计数一个未完毕的写。然后它取得 inode 的锁，以此保证后续读写时其中的偏移量和引用值能够原子性地更新。表面上，`readi`没有对任何磁盘文件进行读写。但由于 inode 的数据被修改了，因此必须调用`log_write`注册修改 inode 的那个 log。幸运的是，由于 xv6 的接口设计非常友好，这些事情都由`readi`帮我们完成了。最后调用`end_op`时，log 将被写入磁盘，然后写入 log header(所谓的 commit)。这时真正的磁盘读写才会开始[4]。所有这些操作的目的是保证磁盘写关于断电或者突然关机这样的意外是原子性的。`end_op`最后擦除 log head。

注意，读写范围是从`va`开始的`PGROUNDDOWN(va + PGSIZE)- va`个字节[5]。其中`va`是当前 VMA 起始地址和真正触发缺页的用户虚拟地址所在页的起始地址中的最大值。如下所示：

```
/* kernel/trap.c:usertrap() */
va = p->vma[i].addr > PGROUNDDOWN(va) ? p->vma[i].addr :
    PGROUNDDOWN(va);
```

这不难理解。读写范围应该是当前页中 VMA 占据的部分，故而要取这两个值中较大的那个。注意到 VMA 对应的文件偏移是`offset`，而读写的起始地址相对开始又偏移了`va - p->vma[i].addr`个字节，因此缺页读应从文件偏移为`p->vma[i].offset + va - p->vma[i].addr`的地方开始。

注意，在目前的实现里，`fork`对内存页是直接拷贝的。因此`MAP_SHARED`有失原本的含义——它们在父子进程中不共享同一个数据副本[6]。为此，我们最后讨论 COW 的实现。

3 Optimization

我们的工作将包括如下几个方面：

[4]假定这个时候未完毕的文件系统系统调用的数目是 0。
[5]`i`表示`va`落入的那个 VMA。我封装了一个专门的函数`withinvma`(kernel/proc.c) 做这件事。
[6]根据 lab 的描述，这里如此实现`MAP_SHARED`是没有关系的。

- COW，解决`MAP_SHARED`以及`fork`效率的问题。
  - 一旦出现多个进程共享一个页面，就必须为每个物理页维护一个引用计数 (伴随地需要修改`kalloc`和`kfree`)。
  - 在`fork`中只是简单地拷贝页表 (修改`uvmcopy`)，同时增加引用计数。
  - 除了 lazy allocation，缺页处理程序还要处理 COW 的情况。通过遍历 VMA 判断，它能判断当前页是否共享。如果该地址不在 VMA 或者一个私有页中，则可认为它是进程私有的。
    - 私有：分配新的虚拟页，把它注册到页表中并拷贝内容。恢复页表的权限位，并释放 COW 页。
    - 共享：此时直接恢复页表内容以及权限位。
- Dirty Page，当`munmap`或者`exit`关闭文件时，对于`MAP_SHARED`的区域，只需要写回真正被修改过的页。

  此时必须检查页表的`PTE_D`位。幸运的是，COW 不会修改`PTE_D`位。另外，lazy allocation 保证没有被分配的页根本不可能被修改，因此也没有必要写回。从而，即便考虑 COW 和 lazy allocation，只写脏页这一机制的实现不会受到影响。这一属性大大降低了实现难度。

4 参考资料

1. *xv6: a simple, Unix-like teaching operating system*

   Russ Cox, Frans Kaashoek, Robert Morris *August 31, 2020*
2. *The RISC-V Reader: An Open Architecture Atlas*

   David Patterson, Andrew Waterman *1st Edition*
3. 现代操作系统 [M]

   A.S.Tanenbaum, H. Bos 著, 陈向群 马洪兵 译, 北京: 机械工业出版社, 2011: 47-95