操作系统课程设计

Xv6实验报告

Lab10 mmap

|  |  |
| --- | --- |
| Name | 朱从周 |
| ID | 2351893 |
| Number | 42028701 |
| major | 软件工程 |



同济大学

Tongji University

1. **环境搭建**
2. 下载VMware虚拟机，并且下载 Ubuntu 镜像并安装 Ubuntu 系统。
3. 启动Ubuntu，安装本项目所需的所有软件，运行：

$ sudo apt-get update && sudo apt-get upgrade

$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu

1. 下载xv6内核源码

$ git clone git://github.com/mit-pdos/xv6-riscv.git

1. 更新镜像源

$ sudo nano /etc/apt/sources.list

$ sudo apt-get update

1. 获取源码

$ git clone git://g.csail.mit.edu/xv6-labs-2021

$ cd xv6-labs-2021

$ git checkout mmap

**1. 实验目的**

本次实验旨在向 xv6 操作系统添加 mmap 和 munmap 系统调用，实现对进程地址空间的详细控制。通过实现这两个系统调用，我们可以实现内存映射文件的功能，包括共享内存、将文件映射到进程地址空间等。这有助于理解虚拟内存管理和页面错误处理的机制。

**2. 实验内容**

**1、添加mmap和munmap的系统调用接口（和前面的实验一样）**

在Makefile中添加$U/\_mmaptest\

**在kernel/syscall.h中添加**

#define SYS\_mmap 22

#define SYS\_munmap 23

**在kernel/syscall.c中添加函数声明：**

extern uint64 sys\_mmap(void);

extern uint64 sys\_munmap(void)

**系统调用函数指针数组中添加**

[SYS\_mmap] sys\_mmap,

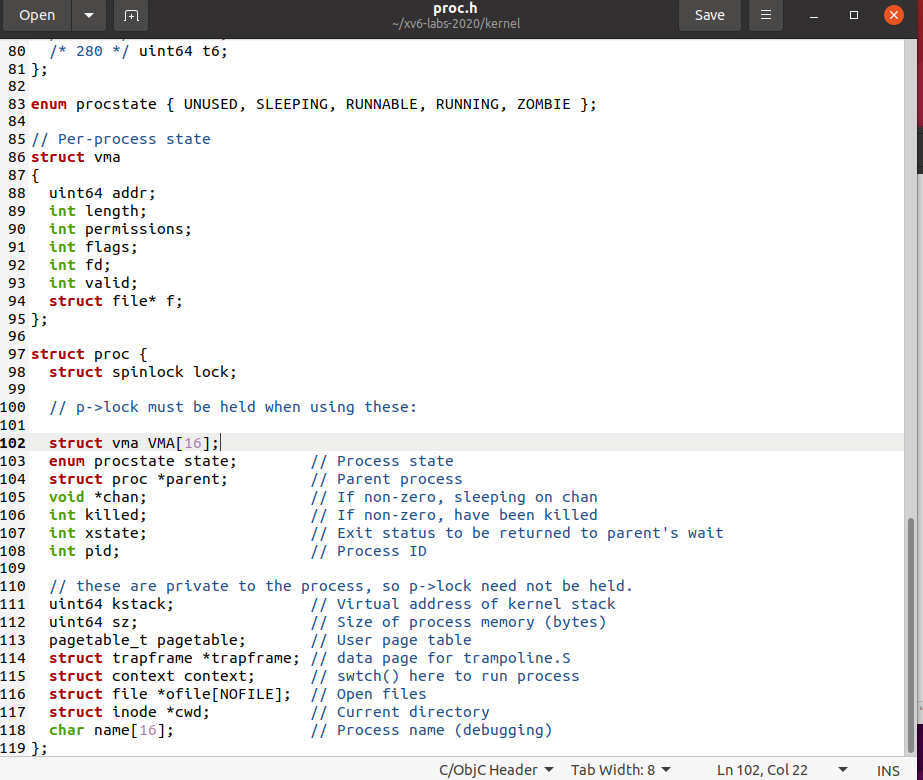
[SYS\_munmap] sys\_munmap,

**在user/user.h中添加函数接口**

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

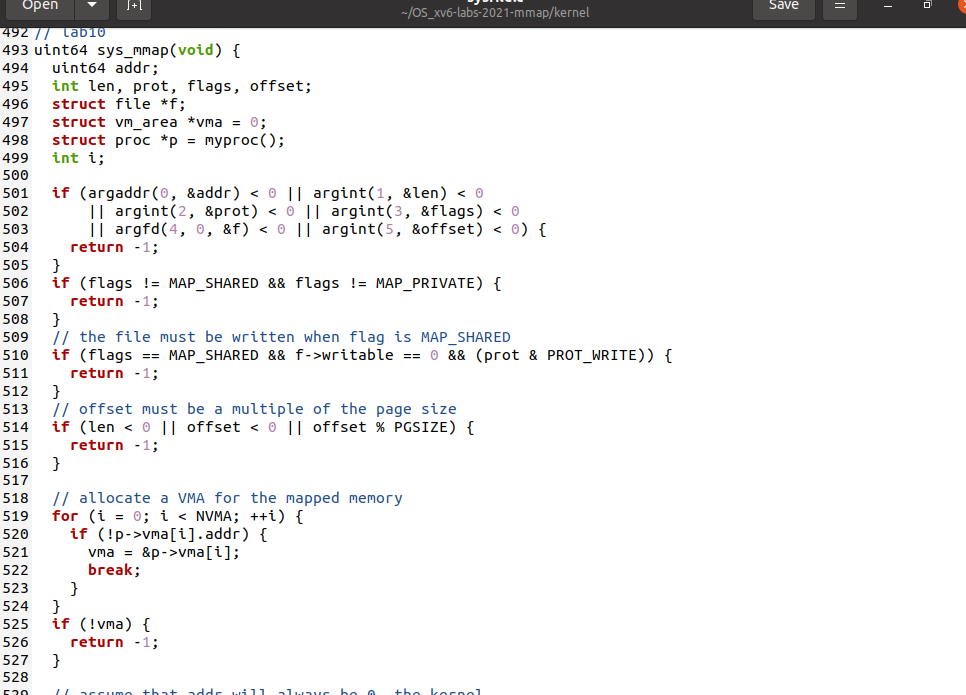
int munmap(void\* addr, int length);

**2、在struct proc中添加字段，用于记录mmap的参数**



**3、实现sys\_mmap函数**

首先读取函数参数，然后判断一下文件的权限和mmap的传入的权限之间的关系是否合法，提高文件的引用计数，分配对应的进程[虚拟地址空间](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9F%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4&spm=1001.2101.3001.7020)，但不分配实际的物理地址空间，并记录在进程的VMA字段中，用于后续usertrap的使用。

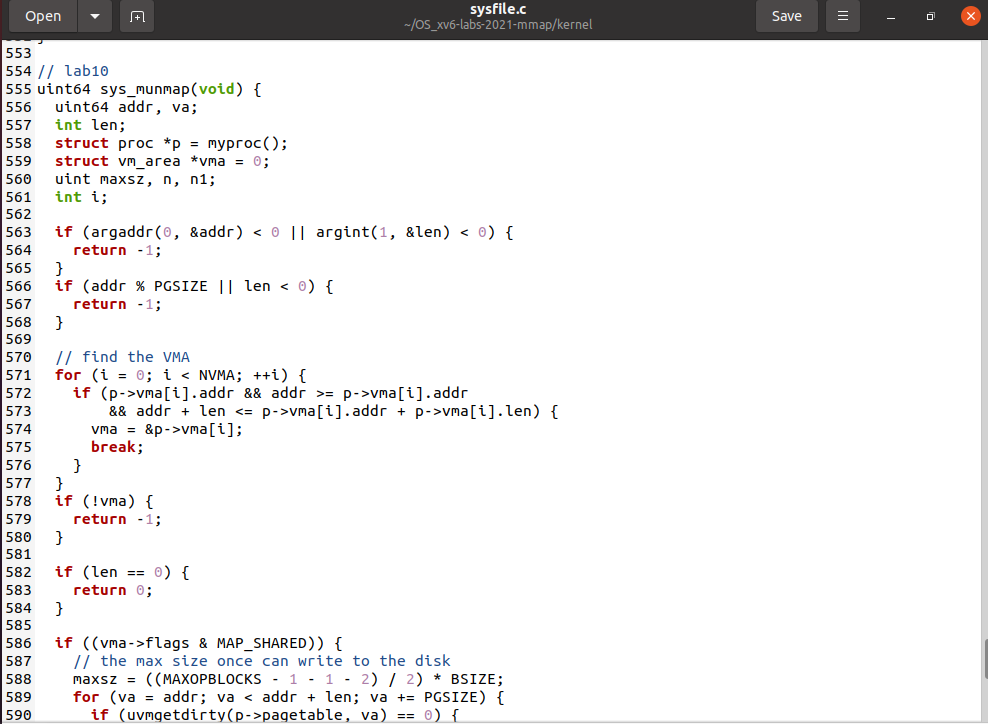


**4、修改kernel/trap.c中的usertrap函数**

读取发生page fault对应的虚拟地址va，根据这个va去查找进程的VMA数组，找到对应的mmap参数，申请一页物理内存，并从对应文件中读取一页的内容写入物理空间，并根据对应的读写权限，加入页表当中（添加映射）。

**5、实现sys\_munmap函数**

如果flag为MAP\_SHARE，那么对于内存的修改要写回文件当中，然后调用uvmunmap，取消所有映射，并释放物理内存，如果释放的是之前mmap的全部空间，要减少文件的引用计数。

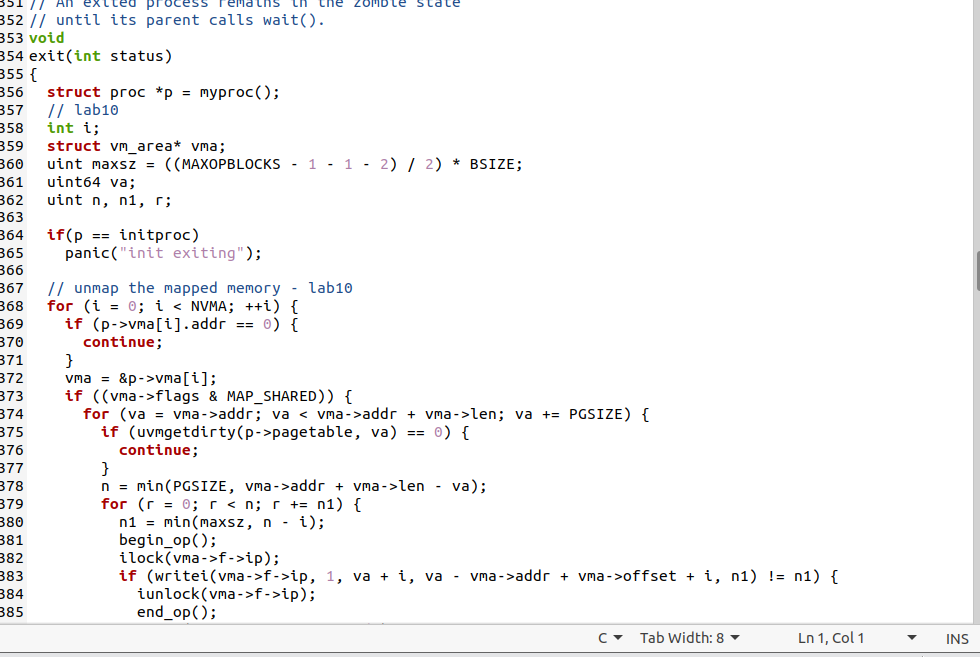


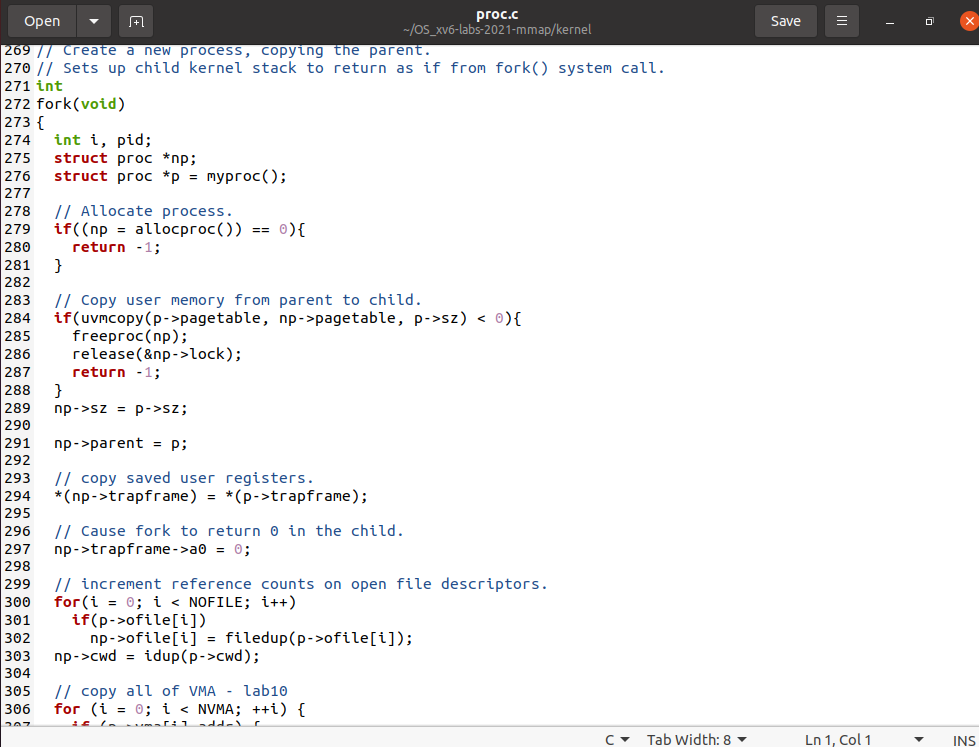
**修改exit**

在进程退出的时候，释放之前mmap对应的空间

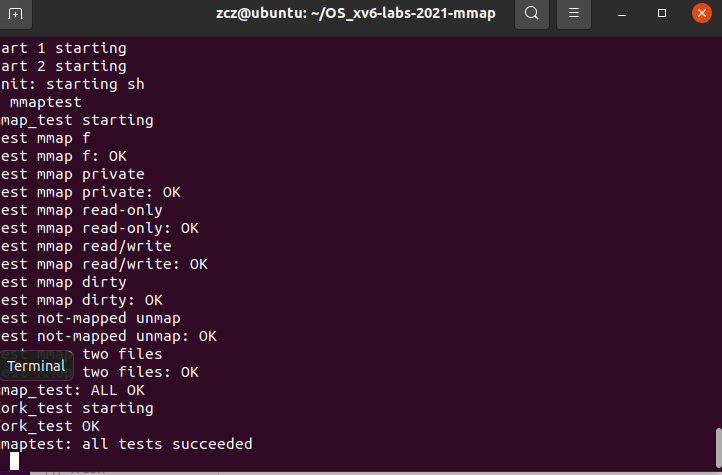
**修改fork**

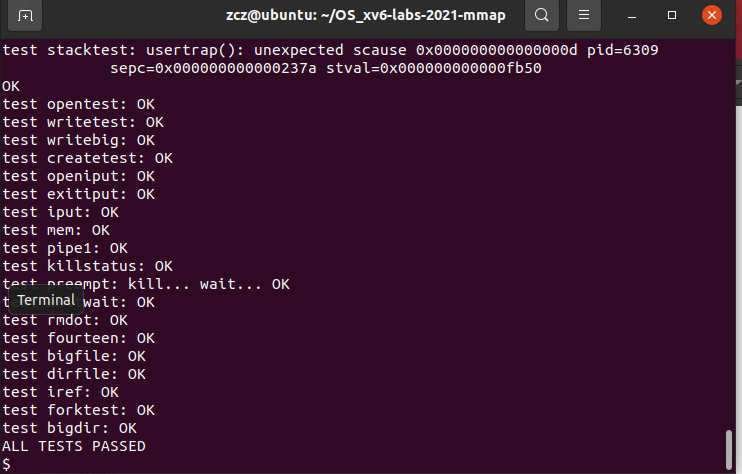
拷贝父进程的VMA数组到子进程的VMA数组。

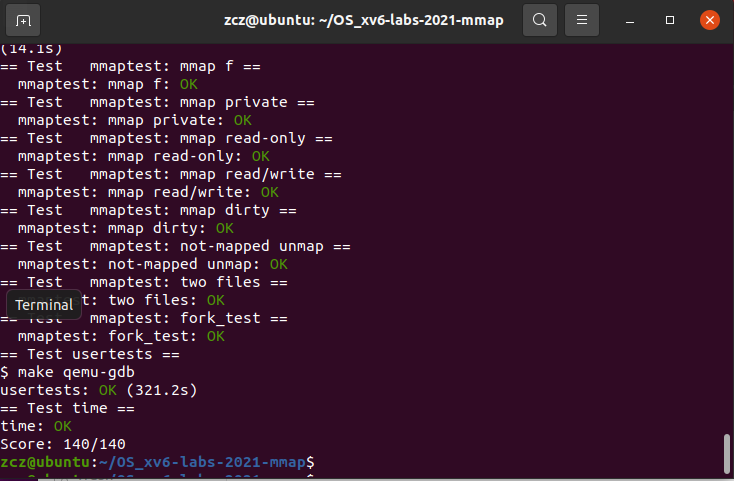




**测试结果：**







**3. 实验分析与心得**

通过本实验，我深入了解了虚拟内存管理、页表操作以及文件系统的交互，同时也熟悉了在操作系统中实现系统调用的步骤和原理。

我成功实现了 sys\_mmap() 系统调用，特别是添加了对映射权限的检查，确保只有可写的文件能够使用 MAP\_SHARED 形式映射。在实现中，我使用了 Lazy Allocation 的思想，仅在页面发生缺页异常时才进行实际的物理内存分配和数据读取。

此外，我还学会了通过合理的标志位设置，实现对脏页的识别和处理，以及根据标志位的设置决定是否写回文件。我还意识到，在实现操作系统的功能时，合理的数据结构和算法设计对于实现的可靠性和性能至关重要。

同时，这一次实验是比较综合性的实验，我能够感受到操作系统的各个部分之间的紧密配合与联系，比如以下的例子：

* **页表与虚拟内存管理：** 实验中涉及到的 Lazy Allocation（延迟分配）机制是通过页表来实现的。在进行文件映射时，并不会立即分配物理内存，而是在页面首次访问时才触发物理内存的分配和内容填充。此间，页表记录了虚拟地址与物理地址之间的映射关系，Lazy Allocation 的实现需要在虚拟地址首次访问时，操作系统根据页表的映射关系将物理内存进行分配。
* **系统调用与用户态交互：** 在实验中，为了支持 mmap 和 munmap 系统调用，需要修改操作系统的系统调用接口。这涉及到用户态和内核态的切换，以及参数的传递和返回值的获取。系统调用提供了用户程序与操作系统内核之间的接口，允许用户程序请求操作系统的服务。在实验中，mmap 和 munmap 系统调用的实现需要正确解析用户程序传递的参数，然后在内核中执行相应的操作，如分配物理内存、建立页表映射等。
* **文件系统与虚拟内存结合：** mmap 系统调用的核心是将文件内容映射到虚拟内存中。这涉及到文件系统和虚拟内存的结合。操作系统需要维护文件的相关信息，如打开的文件描述符，然后通过文件描述符来获取文件内容，并将其映射到虚拟内存的合适位置。这种结合使得用户程序可以直接访问文件内容，而无需显式的读写操作。

实验通过操作系统的各个组成部分之间的紧密联系与配合，这才得以实现一种高效的文件映射机制，展示了操作系统内部不同部分的相互依赖和协作。从页表到文件系统，再到系统调用和虚拟内存管理，每个部分都为实验目标的实现提供了关键支持。这种综合性的设计和实现让我感慨不已，这样精巧的设计使得操作系统能够更好地支持用户程序的需求，也提高了系统的性能和可用性。