|  |
| --- |
|  |
| Lab4/5实习报告 |
|  |

|  |
| --- |
| 姓名：吴悦欣 学号：1900012946  日期：2021/10/28 |

目录

[内容一：实验工作总结 3](#_Toc66996796)

[内容二：遇到的困难以及收获 3](#_Toc66996797)

[内容三：对课程或Lab的意见和建议 3](#_Toc66996798)

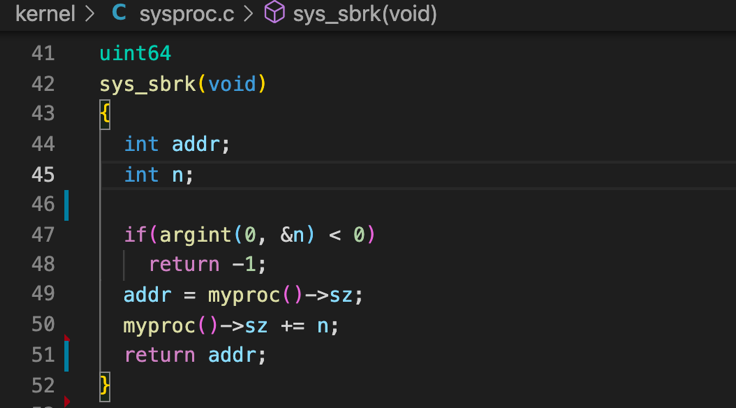
[内容四：参考文献 3](#_Toc66996799)

## 内容一：实验总结

1. Lab4

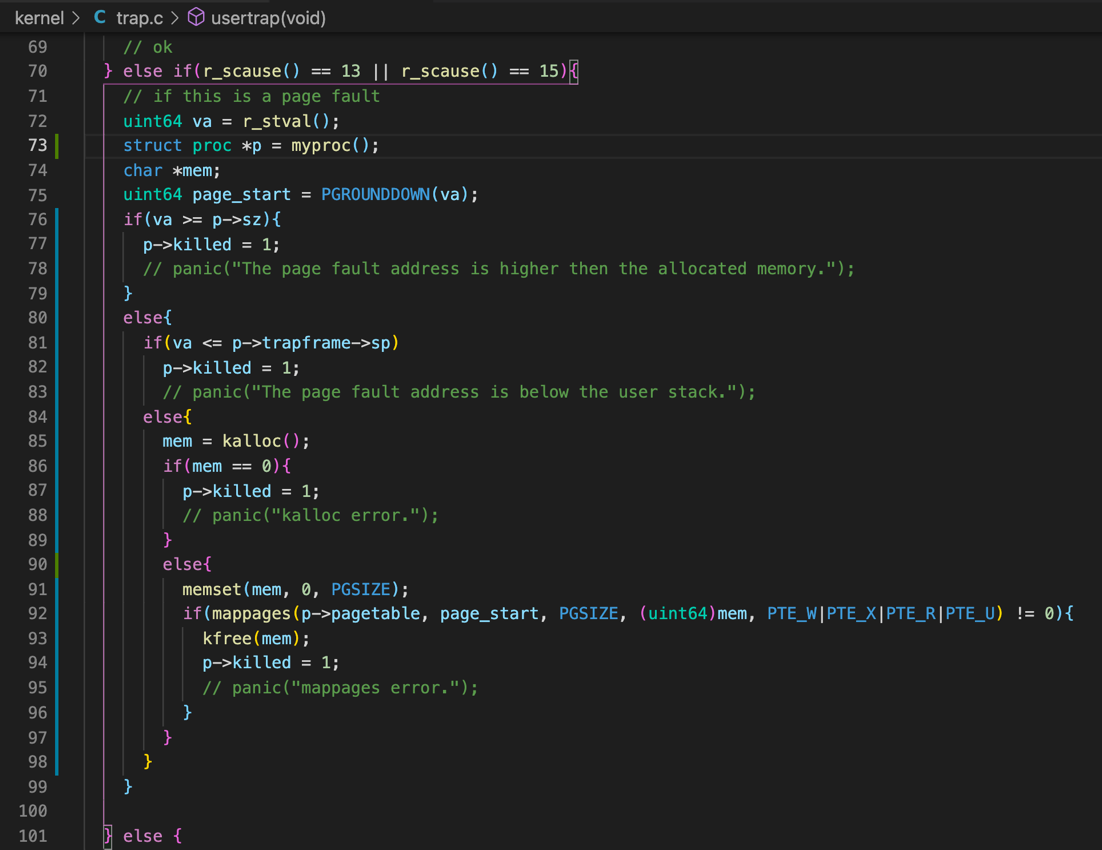
本次实验是关于使用page fault实现懒分配的过程。在xv6课本4.6部分介绍了页故障的类型，在fork后父子进程的写时复制中的作用；实验中则是实现内存的懒分配，有三个部分。

第一个部分是去掉在kernel/sysproc.c的sys\_sbrk函数每次直接分配内存的部分，而是采取先不分配这部分内存，只是在进程的大小中体现实际应当占用空间的大小，等到需要使用这部分内存时产生页故障后再将对应的物理页映射到内存中。这部分的实现很简单，只需要去掉growproc函数，只增加进程的大小。

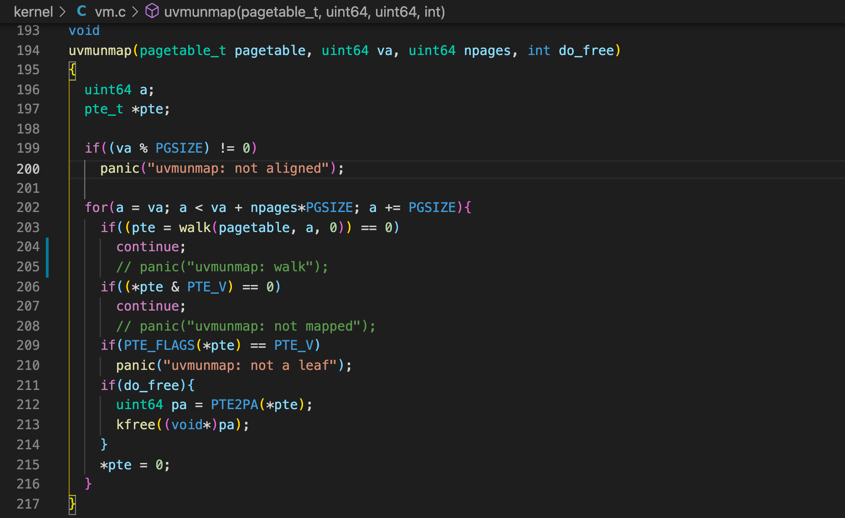


在第一部分完成后，这个系统显然是有问题的，因为并没有分配内存的步骤，在后续的内存的访问中会引发异常。因此在第二部分的工作就是在需要用到某个尚未分配的页的时候，进行内存的分配。由于在使用未分配的地址空间，会产生页故障，陷入内核，并且进入kernel/trap.c中的usertrap函数，而页故障的类型是通过判断在scause寄存器中存有的编号，引发故障的地址则保存在stval寄存器中。由此进行页故障的判断以及页面的分配。

以下是第二部分的代码实现（第三部分的部分实现也已经在里面）



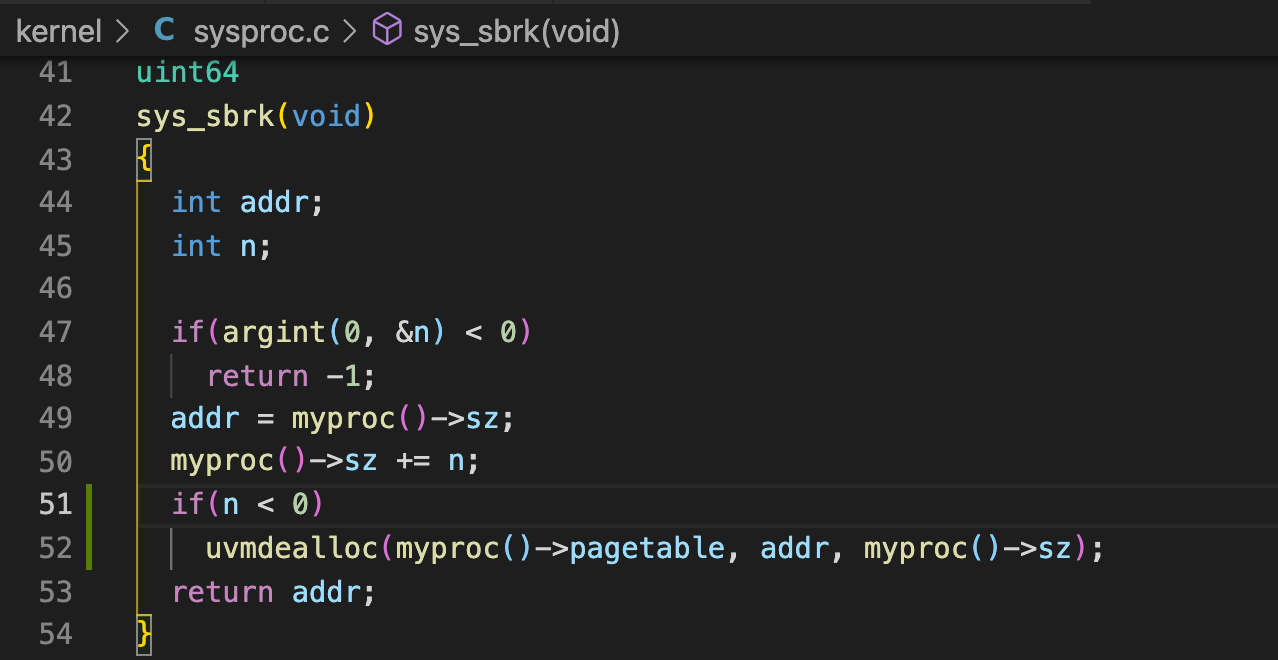
这一部分代码根据提示借鉴了kernel/vm.c中uvmalloc函数的实现，主要的不同是去掉了循环（因为懒分配只需要分配目前使用到的页面），基本和uvmalloc的实现一样。在运行了echo hi指令后uvmunmap函数会panic，因为它会检查是否有页表项，因此需要在kernel/vm.c的uvmunmap函数中取消相关的panic。



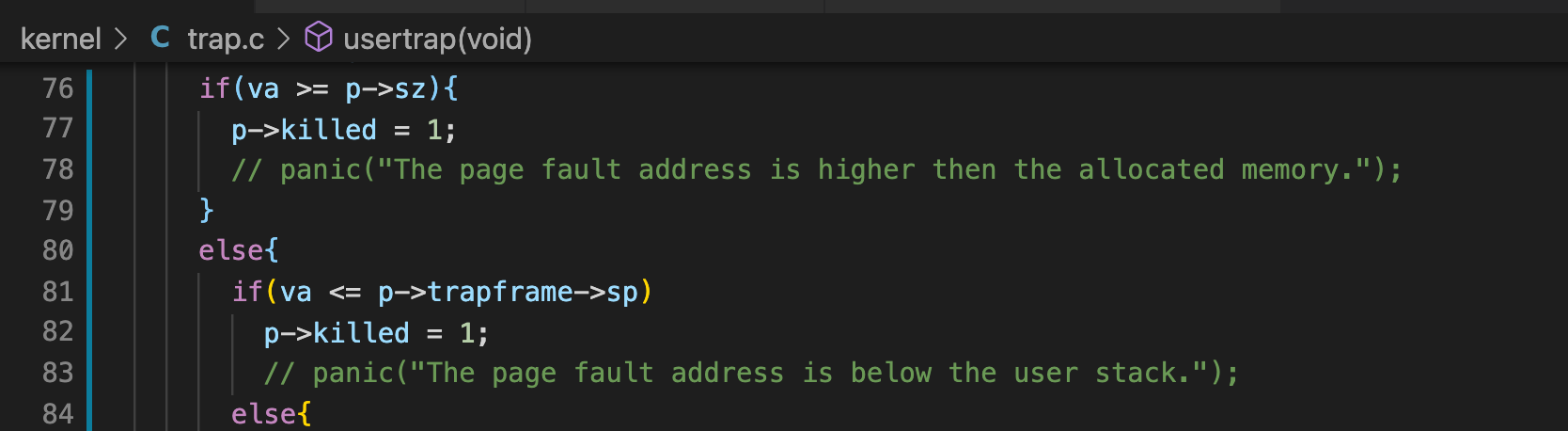
至此，echo hi可以正常运行和响应。

在第三部分中进一步完善了懒分配的机制：

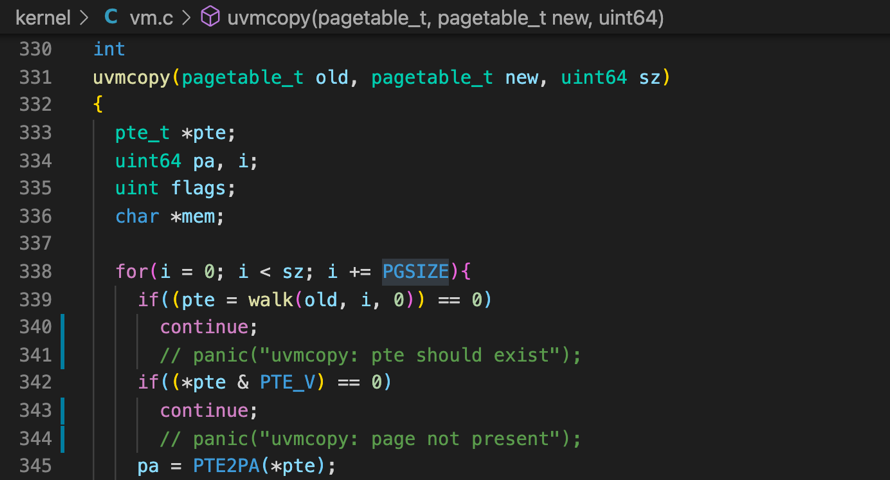
1. 处理sbrk中n为负数的情况，这一步是需要释放n字节的空间，只需要调用kernel/vm.c中的uvmdealloc函数



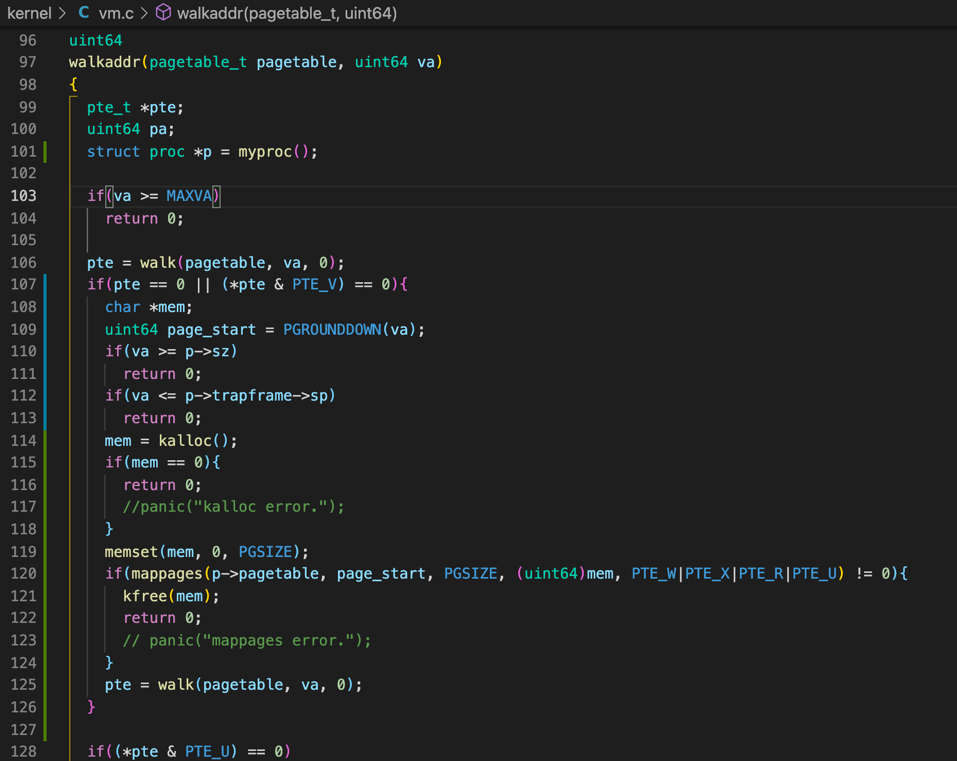
1. 对超过应该分配的空间范围的地址或是非法访问在用户栈之下的页面地址进行处理，杀掉这个进程（下图是在kernel/trap.c的usertrap中的实现）



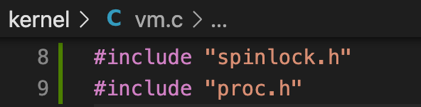
1. 处理进程fork后的内存的拷贝，即在fork时如果拷贝了还没有分配的页给子进程，要防止报错。由于sys\_fork复制内存时是调用的uvmcopy进行的复制，所以以下修改该函数删除panic



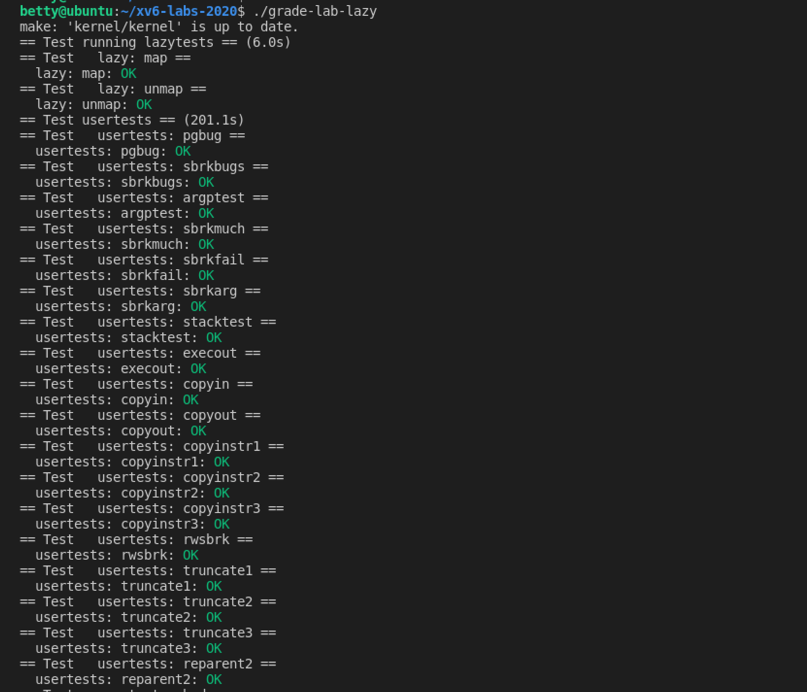
1. 处理系统调用中访问未分配页表地址的情况，并进行页分配。这一部分代码在kernel/vm.c的walkaddr中（被copyout调用）完成，该函数的主要功能是根据虚拟地址返回物理地址，需要将对于pte页表项判断为空或者无效时进行页的分配。



1. 在第三部分的测试的时候出现了“incomplete type proc”的报错信息，因此需要在vm.c中加上头文件spinlock.h和proc.h



至此，所有的测试都可以通过，实验结果如下

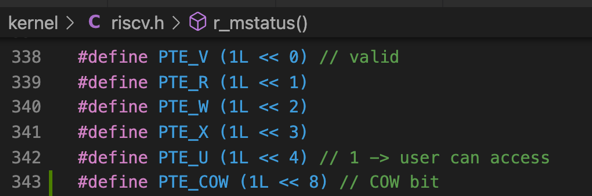


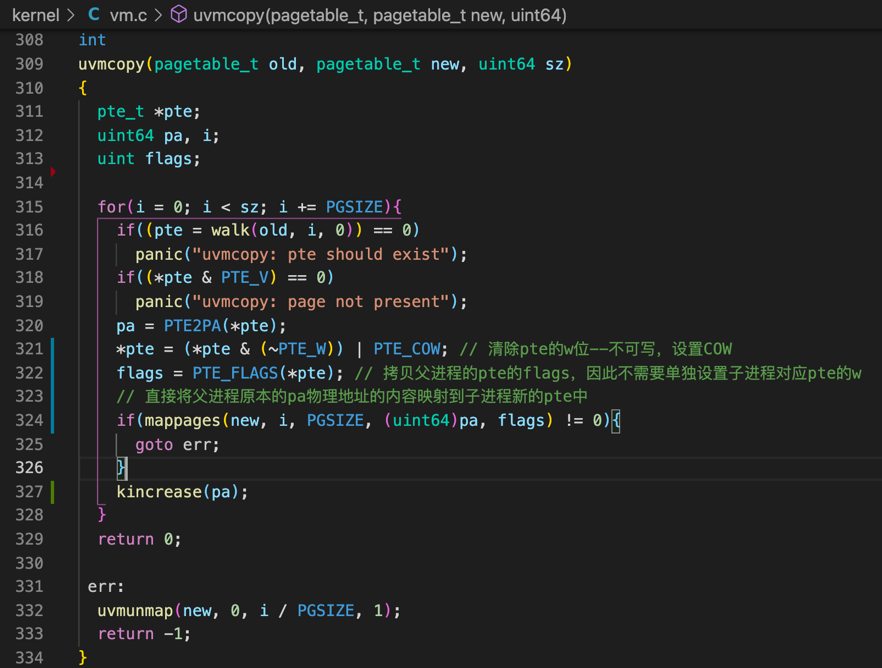


1. Lab5

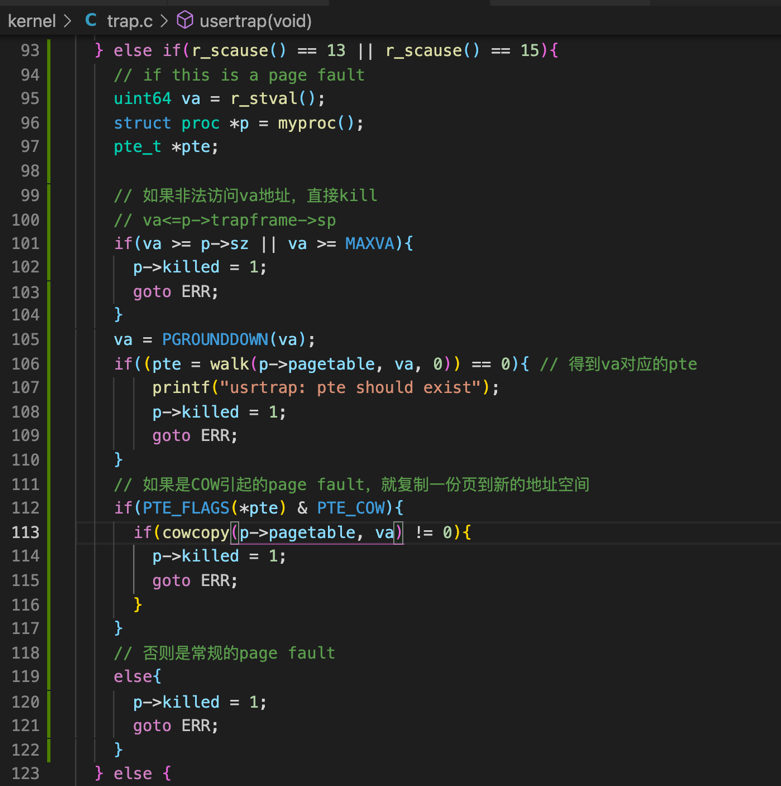
Lab5的目的是实现fork的写时复制（Copy-on-Write），基本的表现是在fork的时候父子进程不同的页表项（地址空间）指向同一个物理地址，并将页表项的标志位改为只读，由此在访问时产生page fault进入异常控制流，才真正复制一份物理地址，从而节省不必要的物理空间开销。具体实现细节如下：

1. 修改proc.c/fork函数调用的vm.c/uvmcpy，该函数原本的行为是将父进程的物理内存完整拷贝一份到内存的另一块空间，并且完成子进程页表项的创建和映射。修改后的行为是直接创建子进程的页表项使之映射到和父进程相同的物理空间上。之后将两个进程的PTE的标志位都修改为只读并标记COW。其中COW的标记位是利用了页表项中的保留位（在ricv.h文件中，是PTE的第8/9位）。kincrease函数是对该物理页的引用数加一。

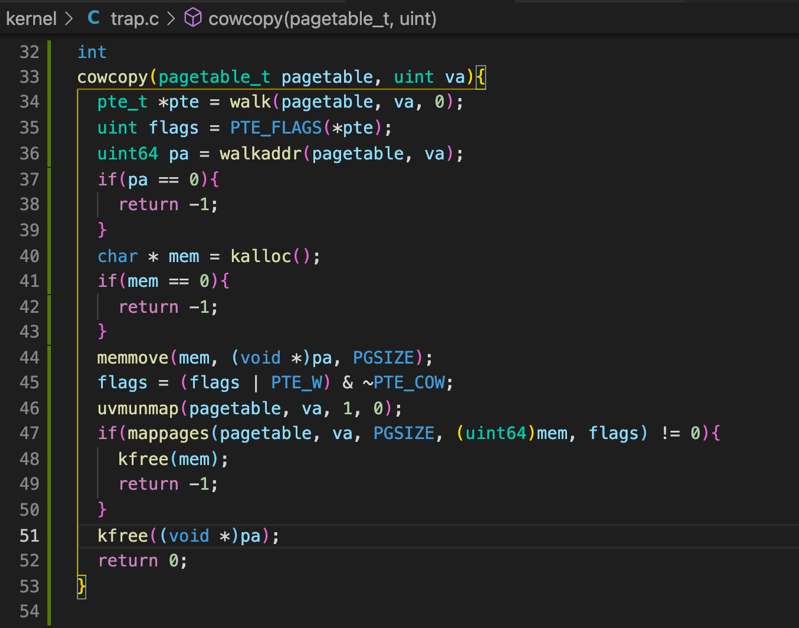




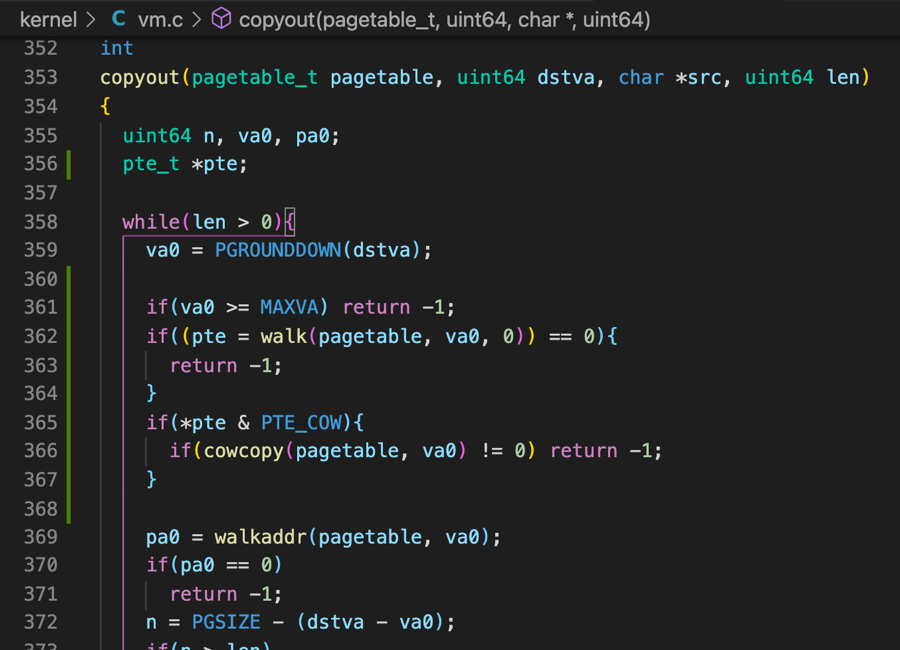
1. 之后更改的是trap.c/usertrap函数，这一函数在上一个Lab中也是类似的作用——处理page fault。类似，这次对于页故障的处理也需要判断地址的合法性（不同点在于这次没有va<=p->trapframe->sp的需求？？？），判断是否是COW造成的page fault之后分配新的物理内存空间并拷贝数据、转移映射。



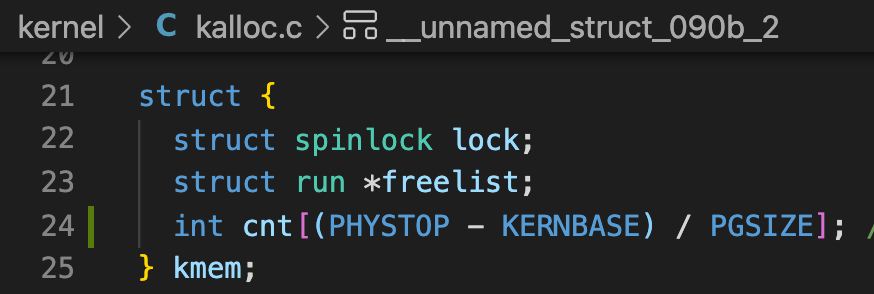
1. 其中trap.c/cowcopy函数是根据页表和虚拟地址具体进行物理内存分配和复制并完成映射的函数，由于在后续还需要用到这一功能，故进行了包装。要注意的是，在将页表项映射到新的物理地址之前，应该将原本的页表项清除，防止重复映射的情况（否则walk会报错remap）。也可以直接使用原本的PTE指针，指向将新地址和标志位合成的地址。这里采用的是调用uvmunmap函数先清除旧的PTE的内容，之后重新进行mappages映射到新的物理地址上。两种方式的不同之处在于，这种实现下，当子进程已经映射到新的物理内存时，如果父进程访问同样的物理地址（在原先的映射下的），父进程不会直接对这个只被自己映射到的地址进行修改，而是同样采取复制一份之后再进行修改的策略。而后者的实现方式可以对于物理页的引用数进行判断之后直接将父进程的页表项的标志位恢复（可写，非COW）。但宏观的表现不会有太大区别。



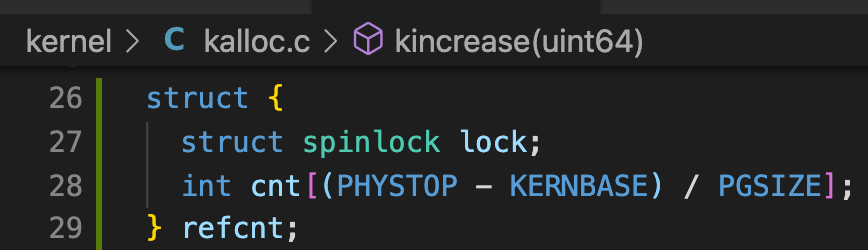
1. 除了用户进程通过异常进行物理内存的分配外，还可能发生的情形是在内核进程中需要访问该物理地址，那么此时可能是禁止中断的，因此不会进入page fault的处理，故要对vm.c/copyout函数进行另外的处理，这里的处理直接调用写好的cowcopy函数即可。一个易错的点是仍然需要判断va0的合法性，这一点很容易被忽略，因为原本的copyout函数中并没有单独的判断语句——是在walkaddr函数的调用中自动检查的。这一点也是在后续debug的过程中让我纠结了很久的点，不得不说Lab3中写的backtrace函数还是比较有用的，在这两个Lab中我都添加了trace查看出错时函数的调用情况来发现一些问题。



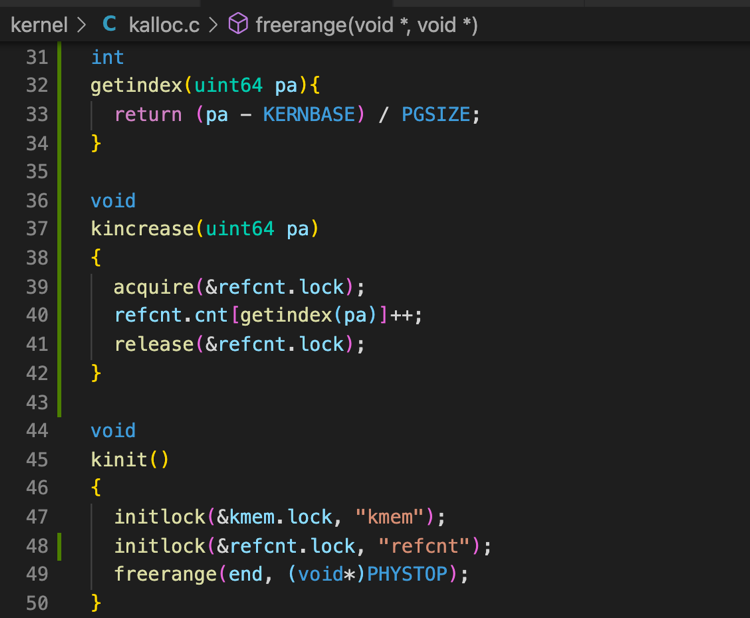
1. 物理页引用数的计数是在kalloc.c文件中实现的，最开始我把相关数组放在kmem的结构体中。

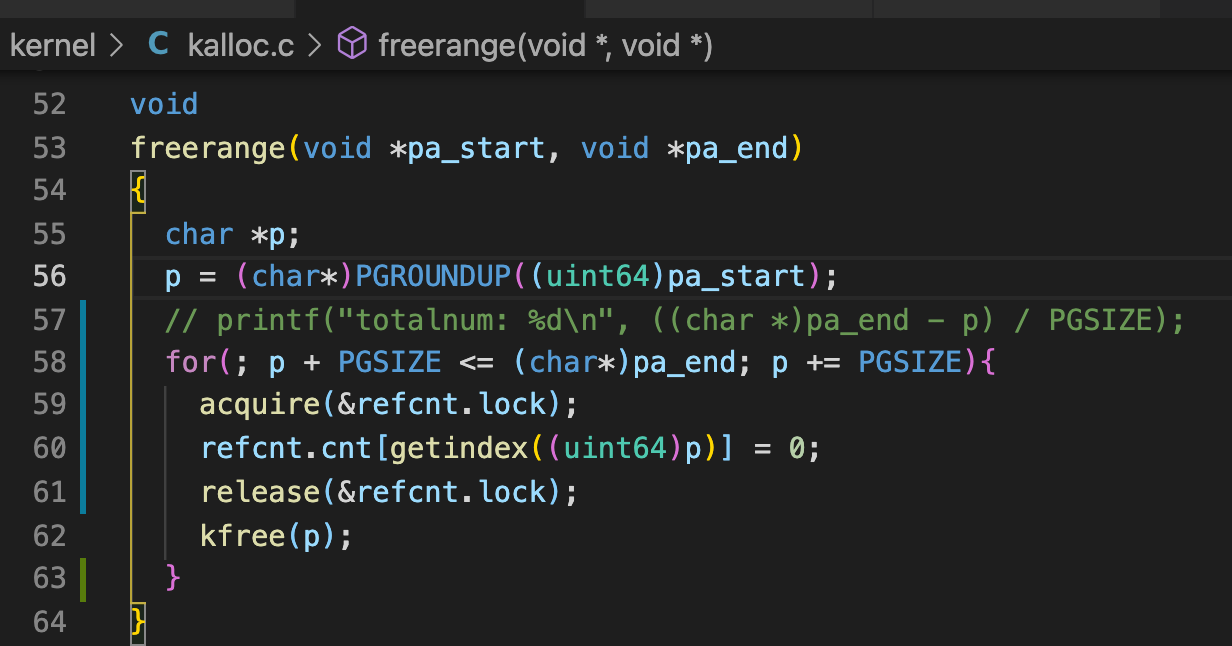


但是因为这种实现方式的互斥关系很容易弄错，所以最后是将cnt数组独立成一个带锁的结构体

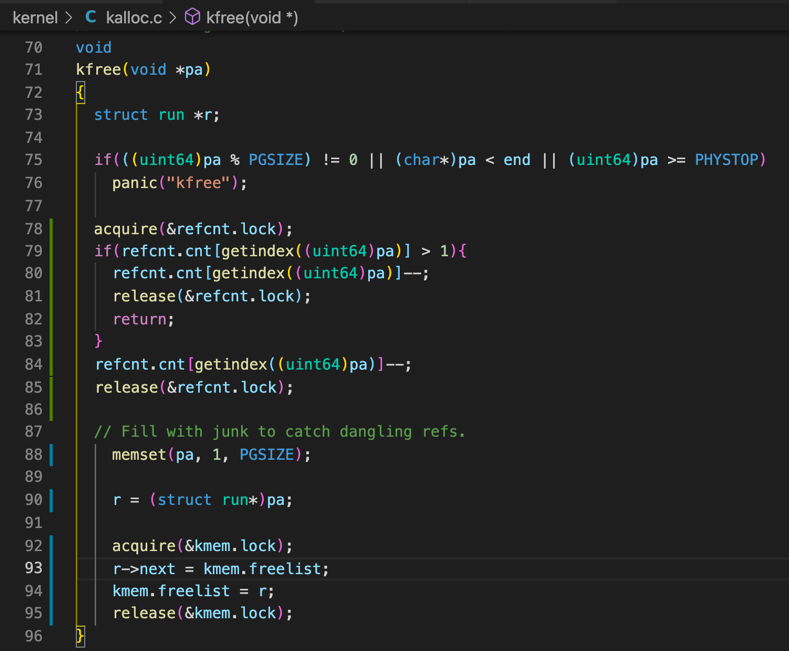


相关的操作包括引用数的修改、初始化以及在kalloc和kfree时的特定操作。

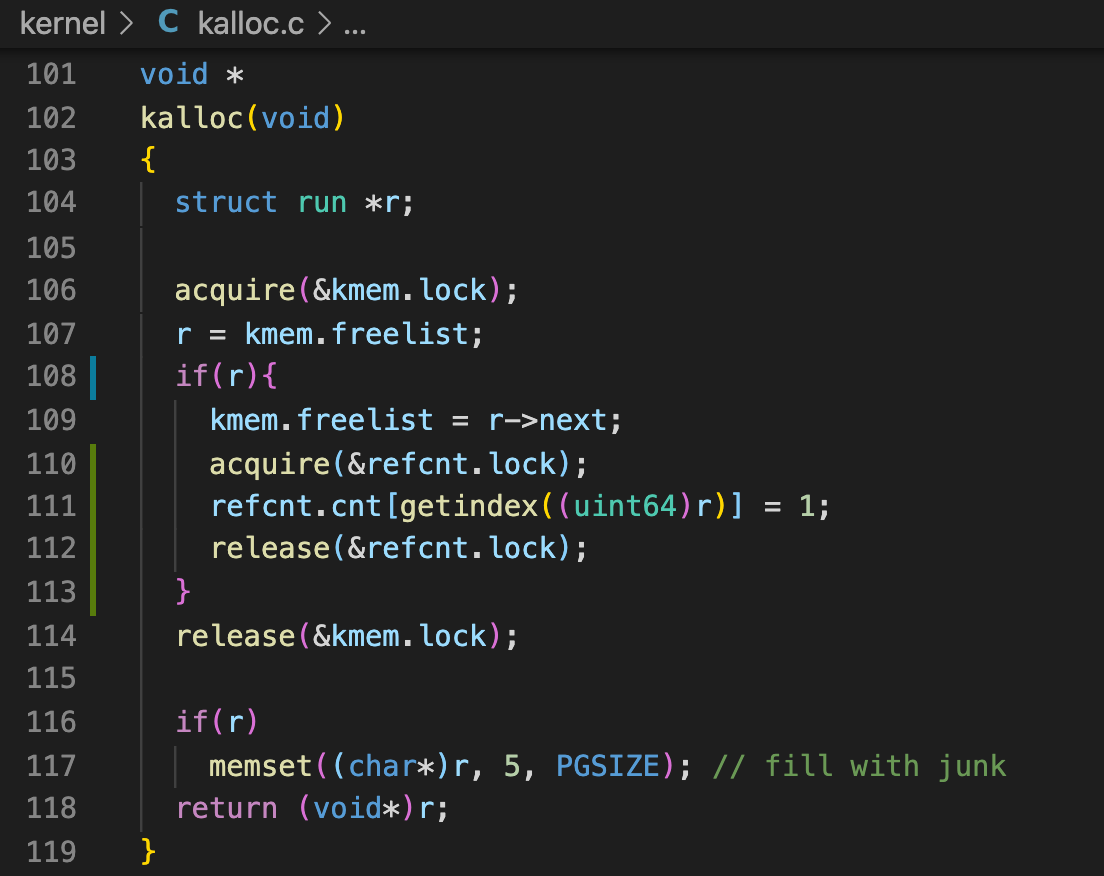




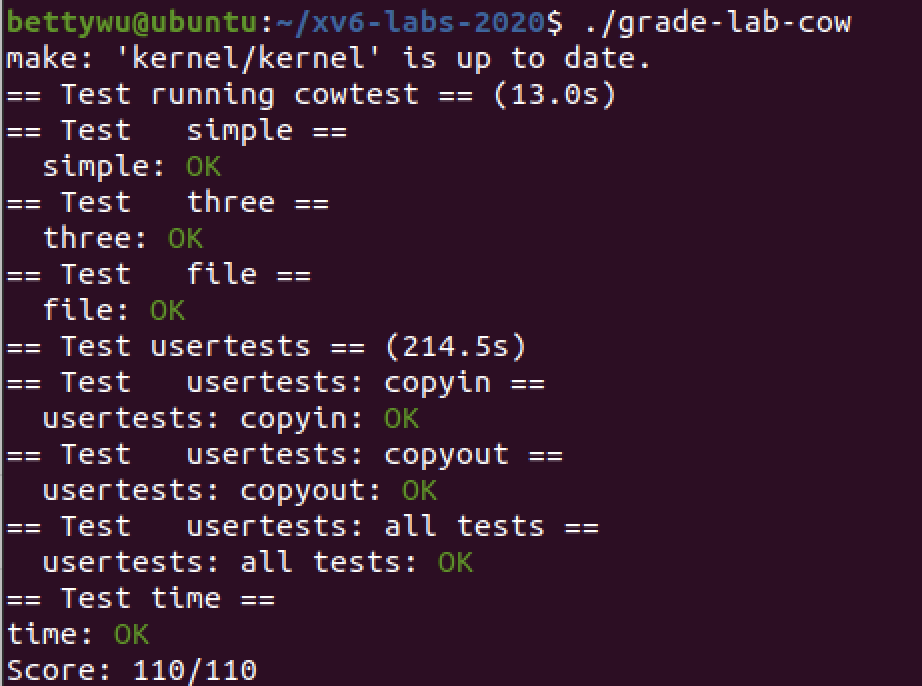
kalloc的时候将引用数设定为1



kfree只有在引用数即将减到0时才真正释放内存



下面是最终运行的结果



1. Lab

## 内容二：遇到的困难以及收获

1. Lab4

在kernel/trap.c的usertrap中对于无效的情况使用panic处理会陷入死循环，并不是要求期望的kill进程的结束方式。在lazytests的时候这里进入死循环还困惑了很久，后来就进到panic函数内部才意识到不应该使用panic。还有在walkaddr的修改中应该符合函数本身的行为，在无效的情况下返回函数值0.

主要的收获是认识到了页面分配的正确时机——在产生用户进程访问内存空间产生page fault陷入内核和系统调用访问未分配的内存空间时。=，进行页面的分配。

1. Lab5

有些部分的处理有一些微妙，包括非法情形的判断和上锁的过程。实际上，在kalloc.c文件中最开始我为减少引用数也写了函数，并且将引用数的数组放在kmem中，然后再kfree调用kdecrease函数的时候发生了互锁的情况。

主要的收获是对于虚拟地址和物理地址的对应关系有了更深的理解，

简要描述自己在实现Lab中遇到的难题和解决方法，以及在Lab实现中的收获与感想。内容要具体，按实际情况撰写。

## 内容三：对课程或Lab的意见和建议

欢迎同学们对本课程或Lab提出宝贵意见和建议。

## 内容四：参考文献

列出参考的文章或者网址等。

注意：在实习报告完成后，请同学们记得在目录部分按右键更新域。