**操作系统大作业2**

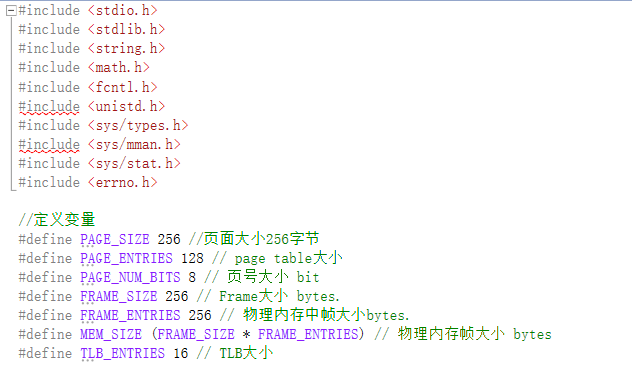
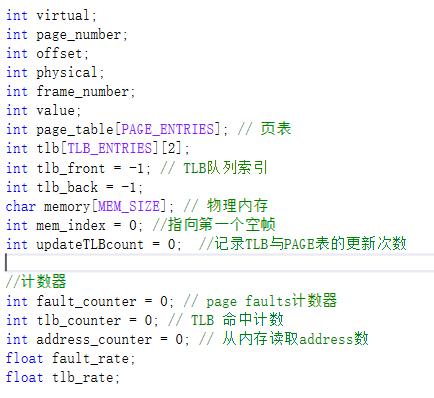
1. **虚存管理模拟程序，40分**

1.1 Chapter 10. Programming Projects: Designing a Virtual Memory Manager（OSC 10th ed.），30分。

保持为vm.c，使用如下测试脚本test.sh，进行地址转换测试，并和correct.txt比较。

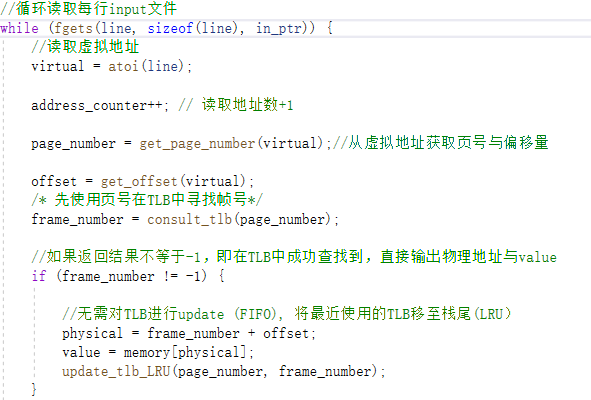
实现TLB的LRU与FIFO代码如下：

基本实现思路即通过一个二维数组维护TLB表并通过不同的算法在对其进行更新，而在实现TLB表的更新算法前，物理地址与虚拟地址的转换则可直接通过页表获得，TLB命中会加快物理地址的转换。

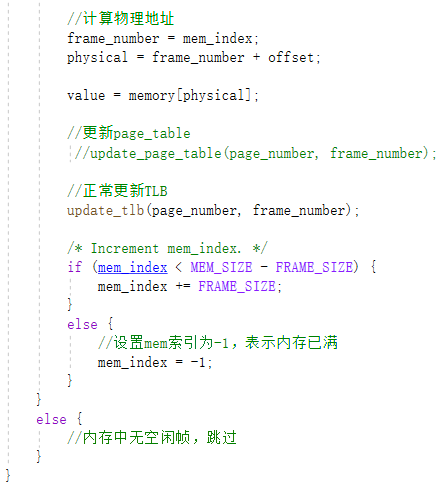


主函数核心部分代码：



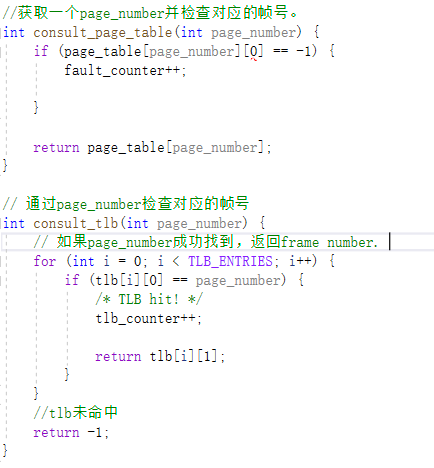




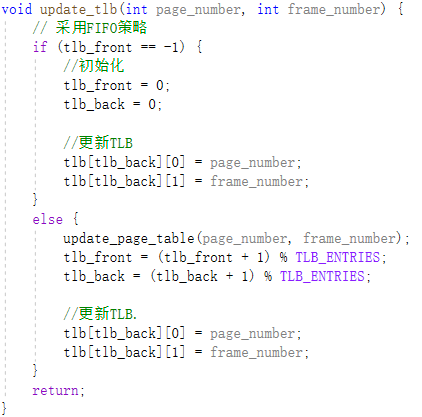


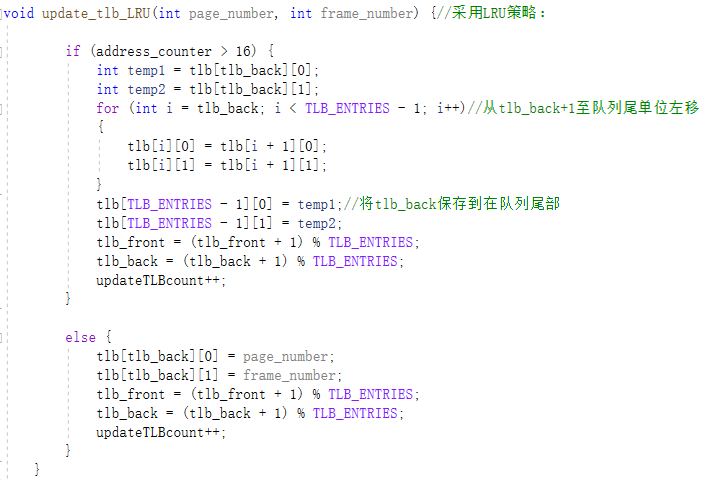


部分的关键函数实现：

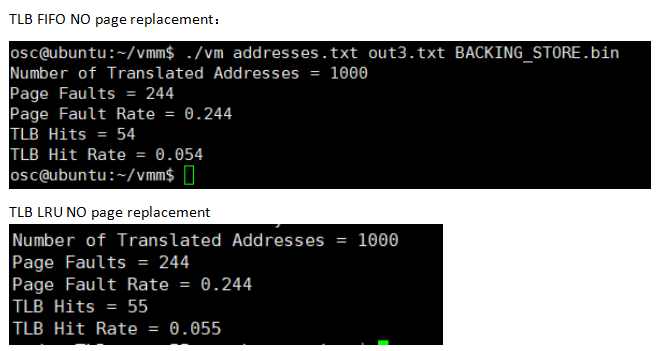


更新TLB函数(先进先出)与LRU函数

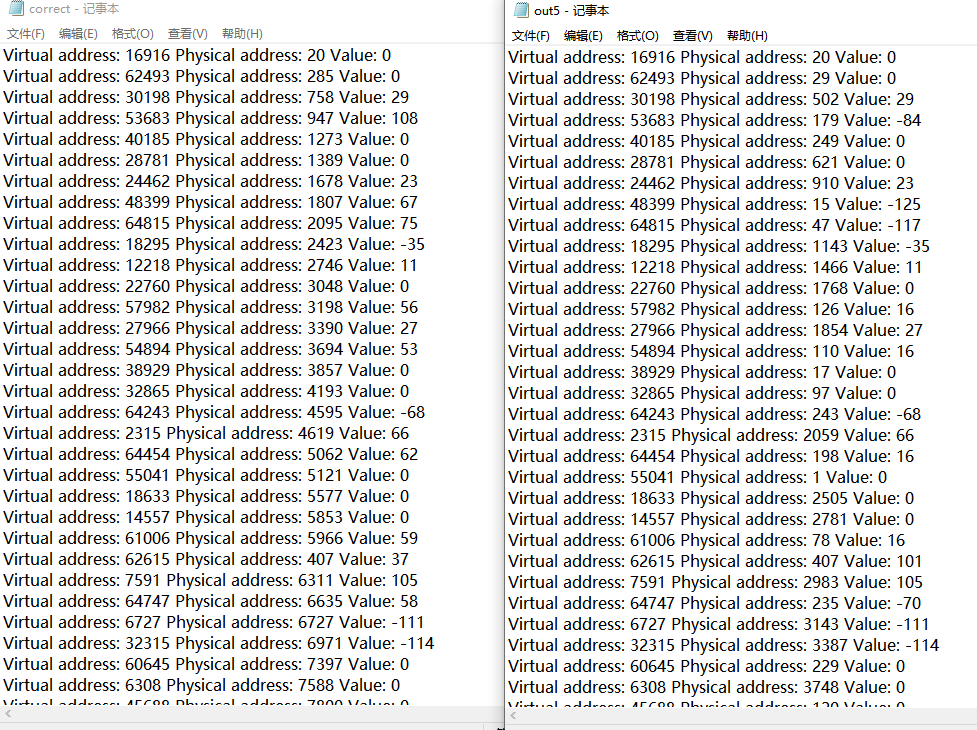




运行结果如下：



TLB击中率由0.054(FIFO)变为0.055(LRU),而转换后的物理地址与correct.txt比较如下图：

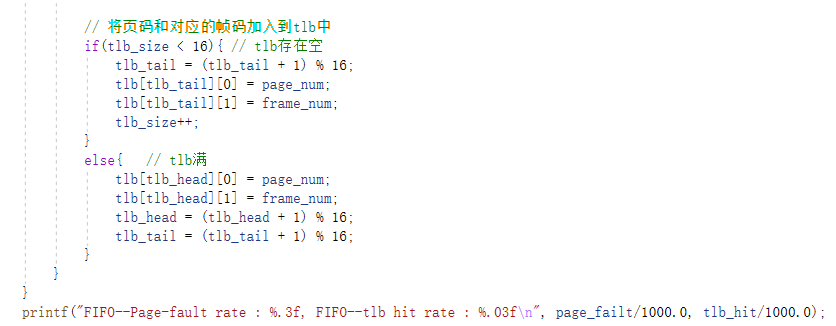


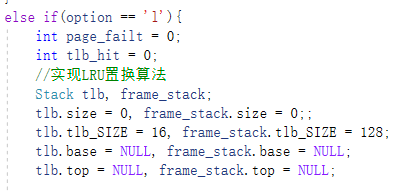
而在之后的页表置换算法的实现时遇到了很多问题，感觉一维数组或者二维数组想要实现PAGE REPLACEMENT十分困难，遇到了大量的BUG,最后选择了另一种完全不同的实现方式，

即直接通过栈和链表来实现对页表的一系列操作。由于之前的实现方式有很多改变，我重新写了一份代码VMM2，将TLB与page\_table的置换算法在数据结构上进行了优化，同时将getopt命令行选择TLB,LRU的操作实现在代码中，页面置换部分实现代码如下：

由于TLB的实现并未有太大变化故不再赘述，下面主要展示page\_replacement中的核心代码：

当发生缺页错误时，

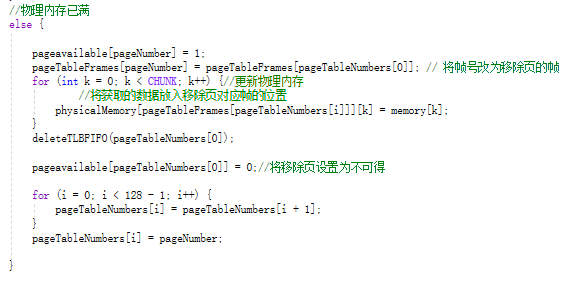


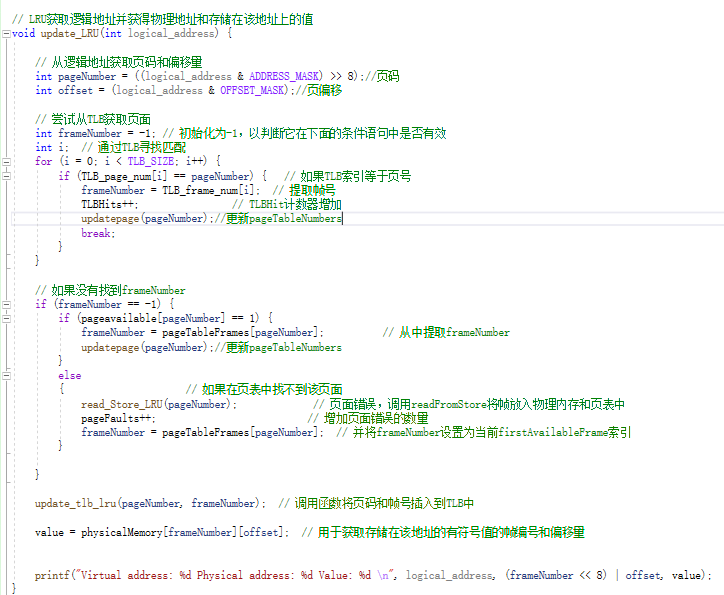


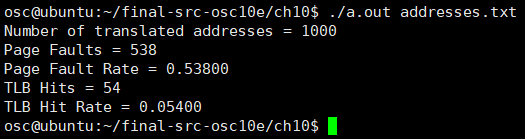
跳过之前已经描述的TLB击中判断，当出现缺页错误时，

从store文件读取帧信息，

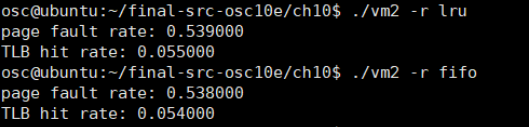




根据LRU算法进行TLB更新与pagetable更新：



运行结果如下图：

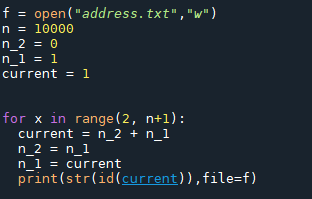


**1.2 设计一个Trace程序**

编写一个trace生成器，记录10000条访问记录，重点是体现内存访问的局部性，并比较LRU与FIFO程序。

trace.py程序通过调用Python自带的id()函数获取地址信息

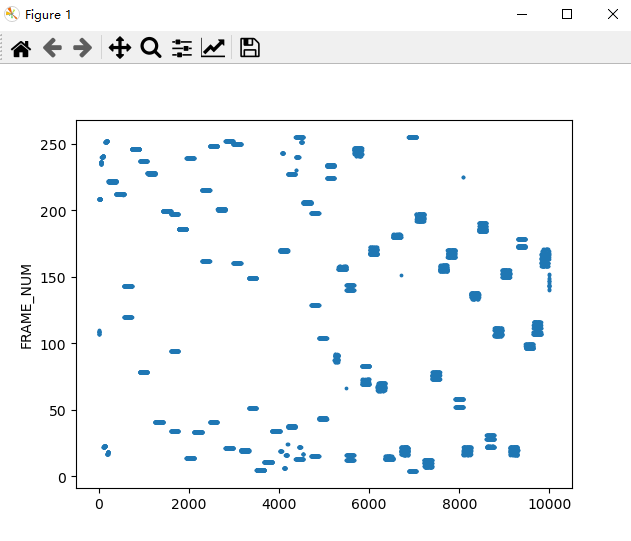
采用斐波拉契数列获得数值的地址，获得其十进制地址信息



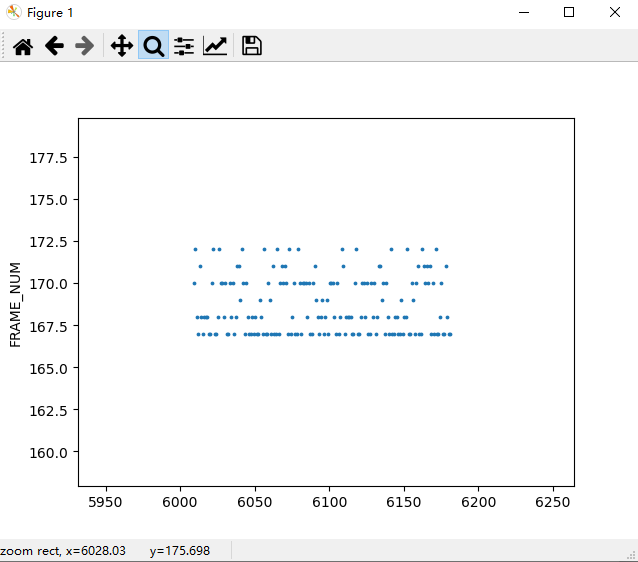
在绘图时将读取的地址文件转换为二进制文件，同时取16位到8位位帧号，打印内存帧号信息如下图：



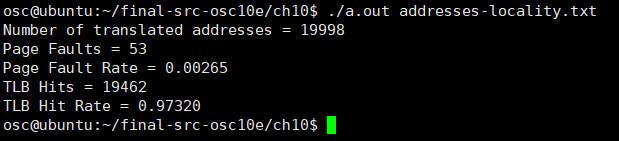
绘制效果如图：



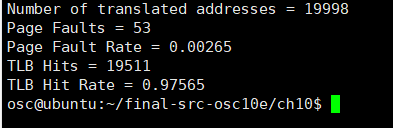
可以看到内存访问时大部分点都聚集在不同的局部区域中，放大后这个现象更为明显：



带入采用之前的FIFO的算法：得到如下结果：

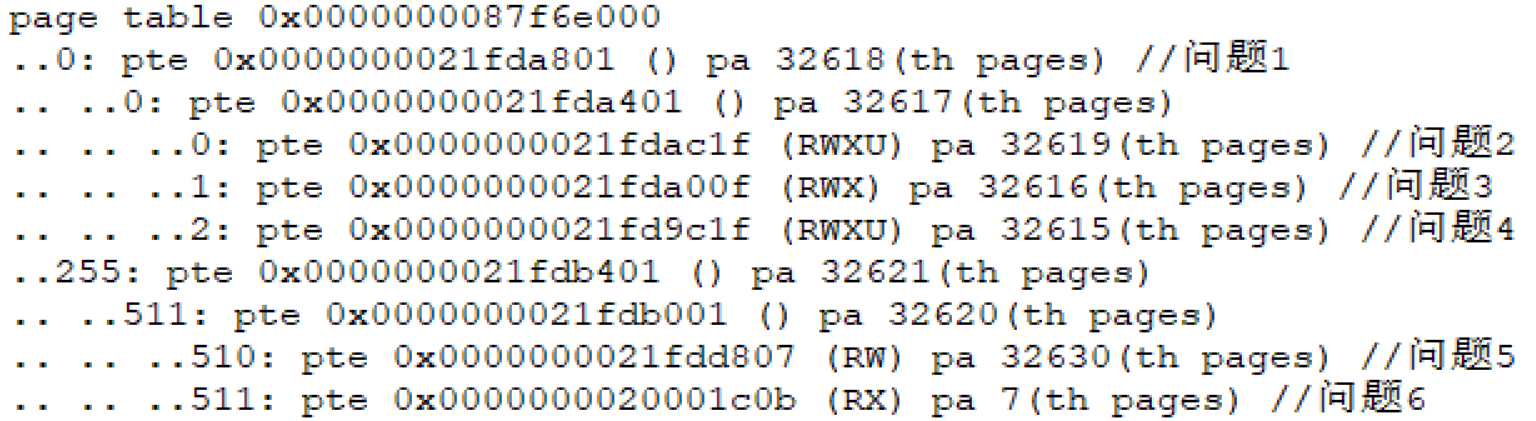


带入采用之前的LRU的算法：得到如下结果：



可以看出LRU与FIFO的TLB命中率都很高，而因为帧数总共为128帧，而由于两种算法内存分布部分相对集中，两种算法的page faults的相近。而LRU的TLB命中率要略高于FIFO算法，证明了算法的正确性，而由于测试函数应该有其他更加能突出内存访问局部性的函数，所以斐波拉契数列的函数选取并不是最佳选择，不过也从定性上与预期实验结果相符。

1. **xv6-lab-2020页表实验（[Lab:page tables](https://pdos.csail.mit.edu/6.828/2020/labs/pgtbl.html)），20分**

****

**图1. init进程的页表内容**

**打印页表代码：**

首先进行页表项权限判定，函数如下：



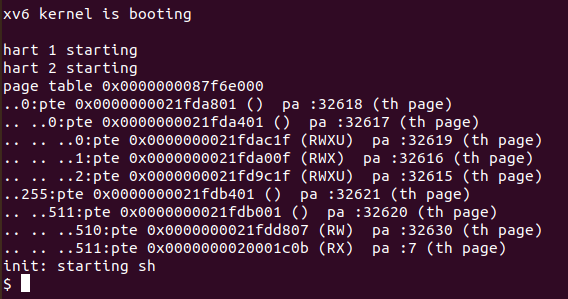
**打印函数：**



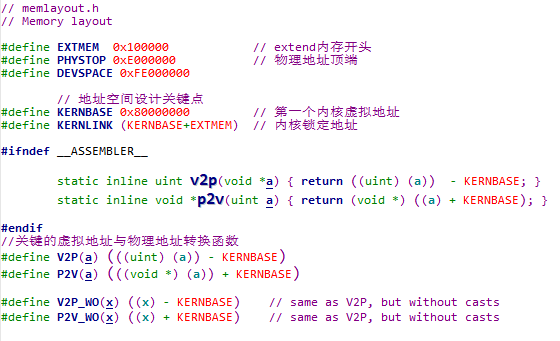
同时在defs.h添加如下函数声明，并在exe.c中增添pid=1时打印页表



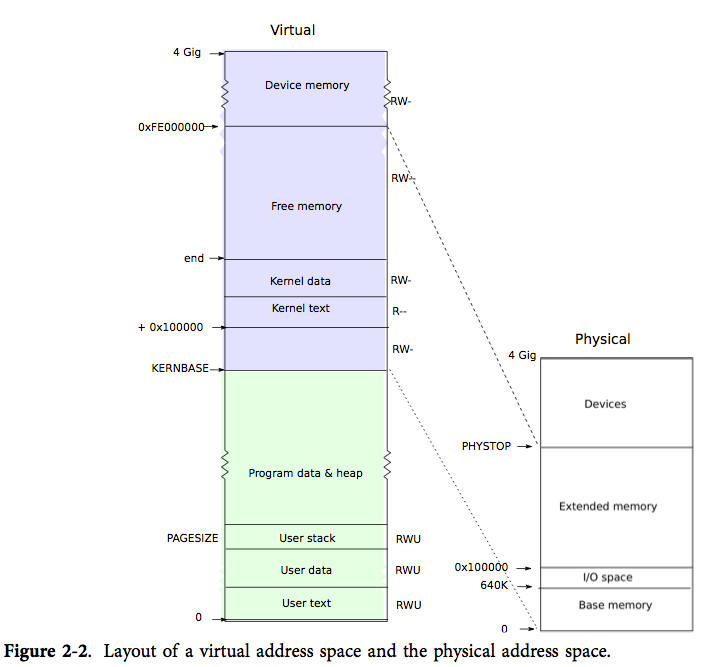
**打印结果如下：**



通过阅读memlayout.h可以看到其操作系统的内存分布的定义：



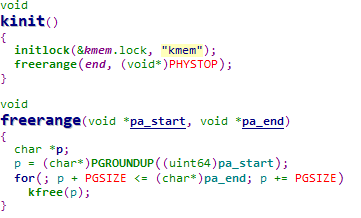
而xv6中整个虚拟内存与物理内存的映射关系如图：



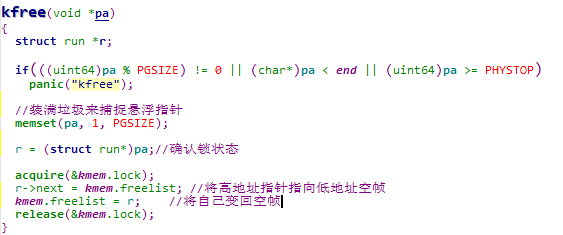
xv6 在每个进程的页表中都包含了内核运行所需要的所有映射，而这些映射都出现在 KERNBASE 之上。它将虚拟地址 KERNBASE:KERNBASE+PHYSTOP 映射到 0:PHYSTOP。具体代码中分配KERNBASE为0x8000000。

**问题1：为什么第一对括号为空？32618在物理内存的什么位置，为什么不从低地址开始？结合源代码内容进行解释。**

答：因为第一对括号为页表创建时初级页表并未提供任何页表权限，如图所示32618页对应其物理内存为：0x0000000087f6a000，而对于为什么从高地址开始分配，则是可以从kinit()函数看出：



内核页表初始化通过freerange()实现，而它的实现虚拟内存分配从END（内核后的第一个地址）到PHYSTOP的区间分配内存,，而这个函数又之后通过调用kfree()函数：将前高地址的页帧的next指针指向了之前已经初始化好的低地址空帧，再把自己设为已经初始化好的空帧，让高地址的指针指向自己，所以形成了地址由上至下的内存释放。Kfree如下图所示：

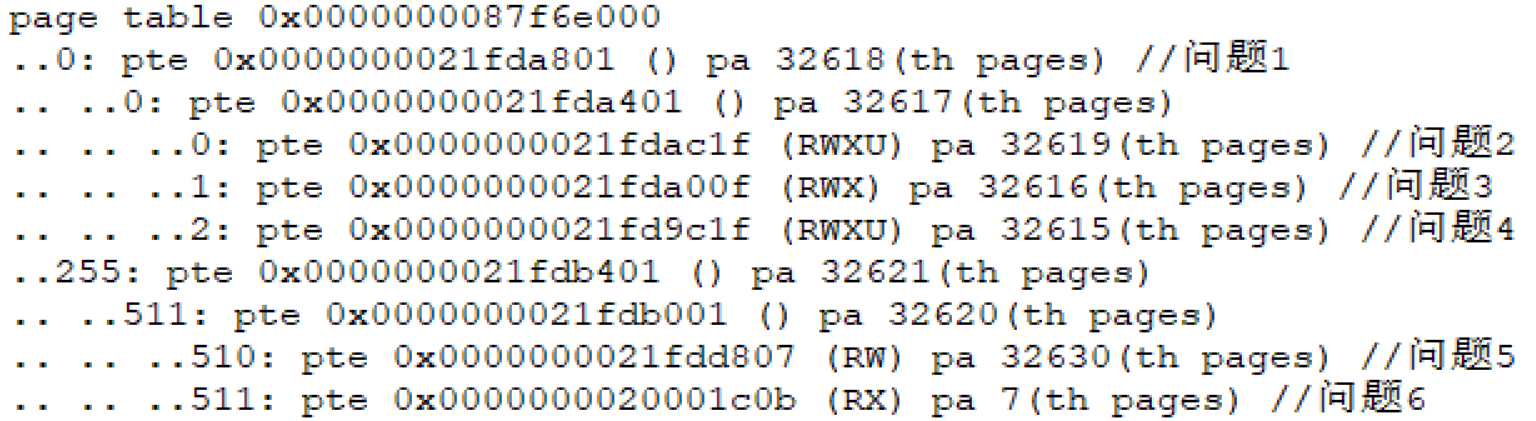


初始化后虚拟地址与物理地址的构建如图[1]：

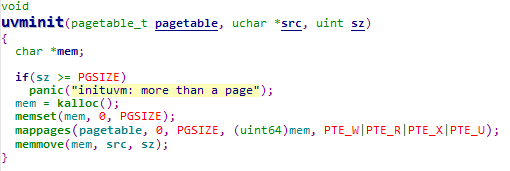


我认为这样做的原因可能是[1]：进程的页表在使用前往往需要初始化，其中必须包含内核代码的映射，这样进程在进入内核时便不需要再次切换页表，进程使用虚拟地址空间的低地址部分，高地址部分留给内核。同时便于不断向下映射内存。

**问题2：这是什么页？装载的什么内容？结合源代码内容进行解释。**

****

**答：**这是uvminit()函数提供的装在用户initcode的页面，如下图：



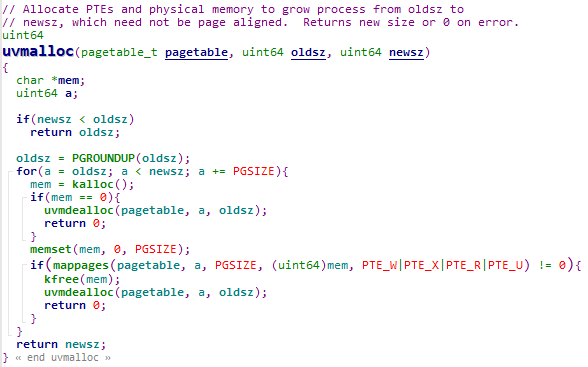
并为其赋予了RWXU权限

**问题3：这是什么页，有何功能？为什么没有U标志位？**

答：我认为该页可能是guard\_page页

**问题4：这是什么页？装载的什么内容？指出源代码初始化该页的位置。**

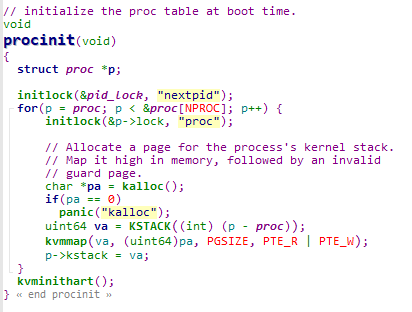
答：这是vm.c中uvmalloc函数生成的页，装载的是进程以后开辟的动态内存，如下源码所示：而如官方注释所言，uvmalloc可以范培不需要页对齐的物理内存和页表项给进程，



**问题5：这是什么页，为何没有X标志位？**

答：这是proinit()函数为内核栈生成的页面

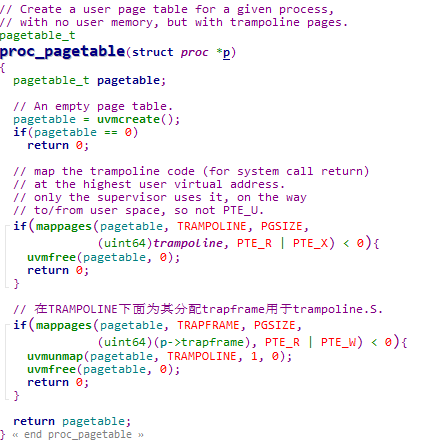
源码如图所示：



由源码注释可看该页放在页表高处，同时隔了一个guard\_page来保护，该页（没有X标志位）的原因是它就是用来分割堆和栈的，通过禁止执行来起到分割效果。

**问题6：这是什么页，为何没有W标志位？装载的内容是什么？为何这里的物理页号处于低地址区域（第7页）？结合源代码对应的操作进行解释。**

答：这是trampoline页,装载内容为uservec即user trap开始的地方，装载内容是进程初始化时已经设定好的，所以不应该被可写。



7

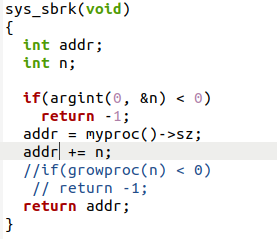
.

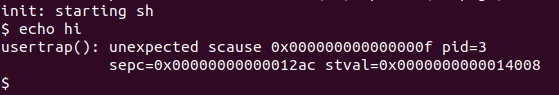
1. **xv6-lab-2020内存分配实验 ([Lab: xv6 lazy page allocation](https://pdos.csail.mit.edu/6.828/2020/labs/lazy.html))，40分**

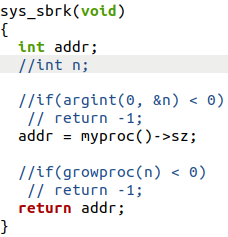
3.1 完成Lazy allocation子任务，要求echo hi正常运行

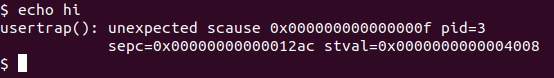
答：尝试了如下变量：

尝试将addr递增，出现如下错误：

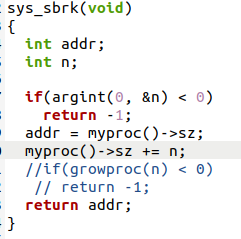


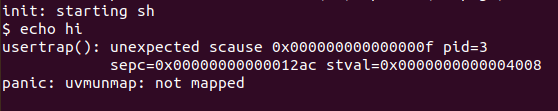


将argint判断消除，错误结果如下：



而将sys\_sbrk()里分配内存的部分删去，仅仅修改myproc()->sz的大小，则可以出现图中关于未分配内存（not mapped）的报错。

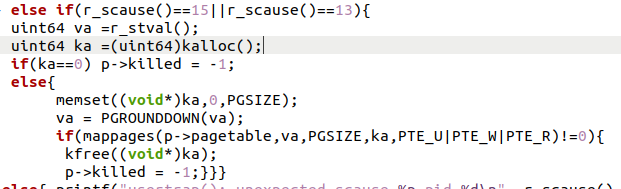




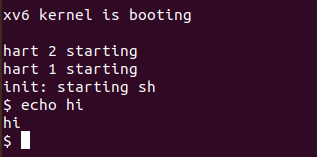
而要解决这个问题，根据问题的提示，我们可以在usertrap函数里，检查r\_scause()函数返回值是否为15或者13，然后通过r\_stval函数获取出现问题的虚拟地址

实现如下（代码中的英文注释都为源码自带）：

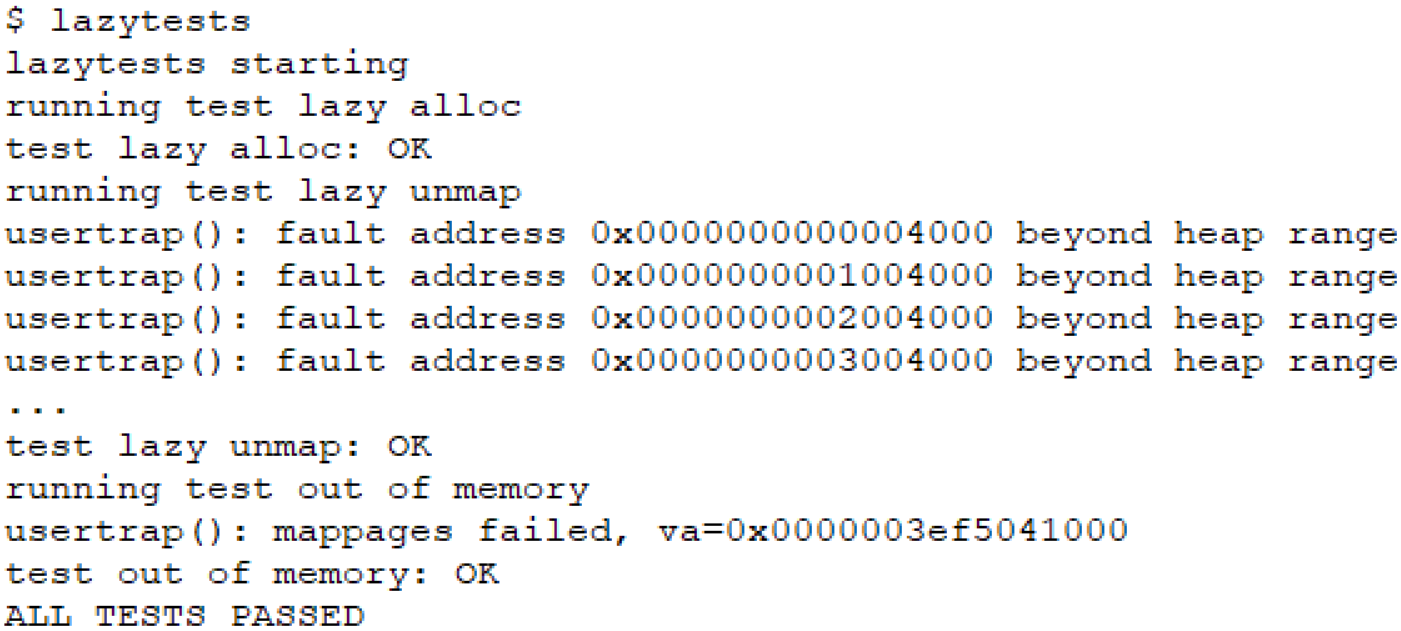
参考vm.c中uvmunmap中对于内存的分配方式，进行内存分配



同时忽略uvmunmap的报错信息后，echo hi 正常运行

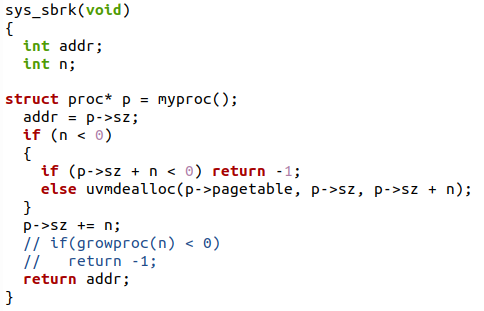


3.2 完成Lazytests and Usertests子任务。对于Lazytests，要求屏幕输出如下图所示；对于usertests任务，要求通过所有除sbrkarg之外的测试。给出运行截屏。

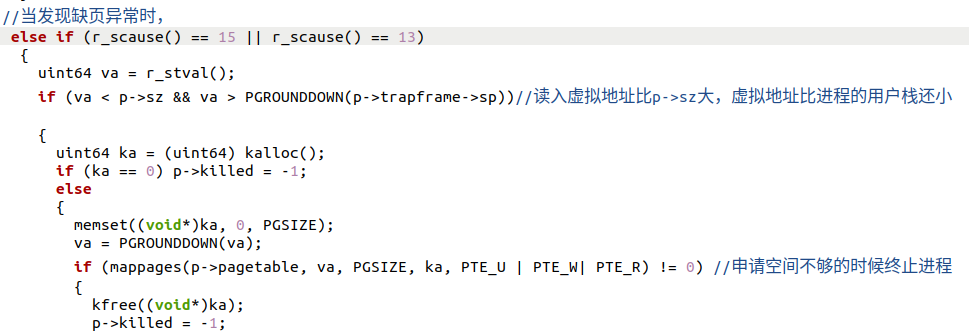


Lazytests实现如下：

在sys\_sbrk（）增加对于n<0的处理，即减小分配内存的情况

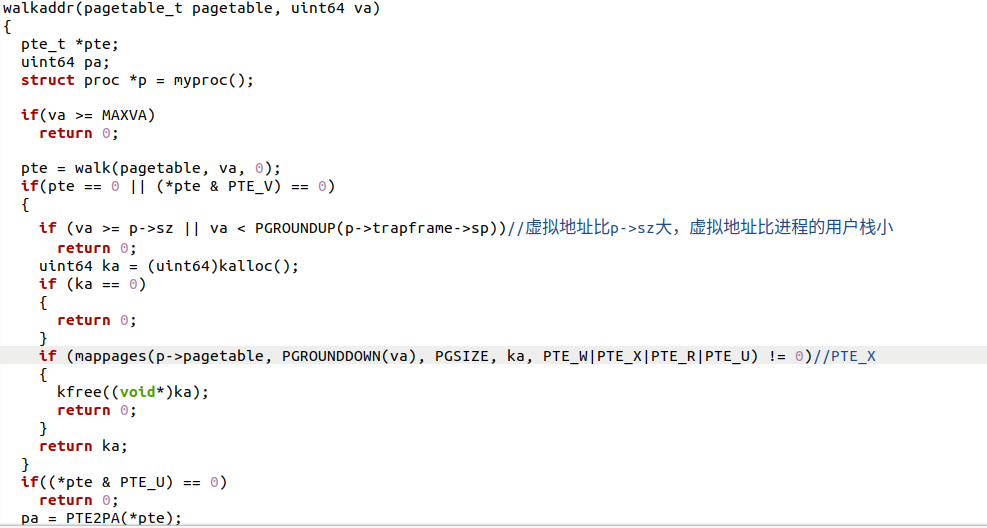


在usertrap中进行改进，增加终止判定条件

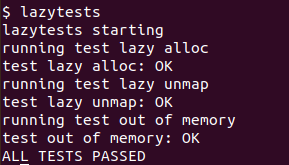


而当系统调用时，得到地址如果找不到相应的物理地址时需要，添加相应物理地址映射到page table里，vm.c中的walkaddr函数可以将虚拟地址转换为物理地址

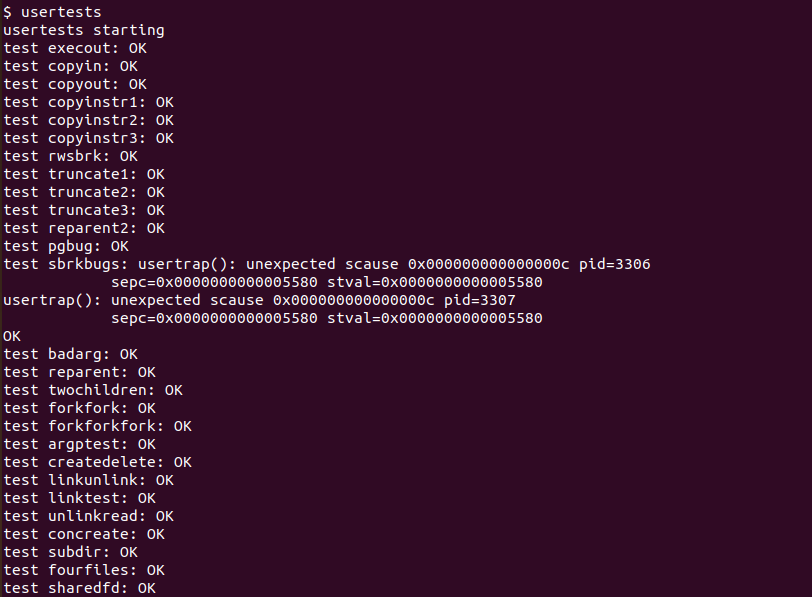
在原代码基础上增加了如usertrap中添加的判定条件，同时增加了可调用权限PTE\_X,

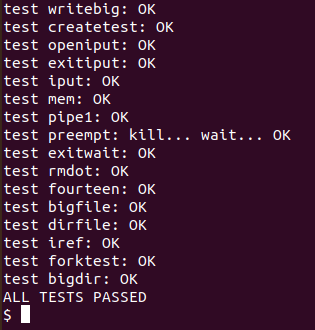


修改后可以完美通过lazytests



而对于usertests，除sbrkbugs之外都成功通过运行





总结：本次实验涉及了操作系统有关虚拟内存管理，内存分配等很多复杂的问题，其中充分考验了我们对操作系统基本概念的理解和掌握实现。而本次实验也还有很多可以改进完善的地方，希望借此机会，加强有关概念的认知，提高编程水平，为以后的相关学习研究打下基础。