# 任务一：打印页表

1、任务

格式化打印三级页表中的有效pte。

2、思路

递归遍历三层页表，可以参考freewalk的写法。原来的过滤目标是(pte & PTE\_V) && (pte & (PTE\_R|PTE\_W|PTE\_X)) == 0，现在设定我们过滤的目标是pte & PTE\_V，这个过滤条件保证了最终输入的pte只有我们感兴趣的valid pte，也就是正在占用的pte。

另外，uint64 pgt\_pa 可以直接转为 pagetable\_t 结构，这为寻找下一级pgt提供了简洁的代码。

3、代码

*// vm printer*

void vmprint\_aux(pagetable\_t *pagetable*, int *level*)

{

  int tab\_wide = (level+1)\*2;

  for(int i=0; i<512; i++)

  {

    pte\_t pte = pagetable[i];

    if(pte & PTE\_V)*// check valid*

    {

*//tab*

      for(int j=0; j<tab\_wide; j++)

      {

        if(j%2==0) printf(" ");

        printf("|");

      }

      printf("%d: pte %p pa %p\n",i, pte, PTE2PA(pte));

      if(level < 2)*//total 3 level*

      {

        uint64 child = PTE2PA(pte);

        vmprint\_aux((pagetable\_t)child, level + 1);

      }

    }

  }

}

void vmprint(pagetable\_t *pagetable*)

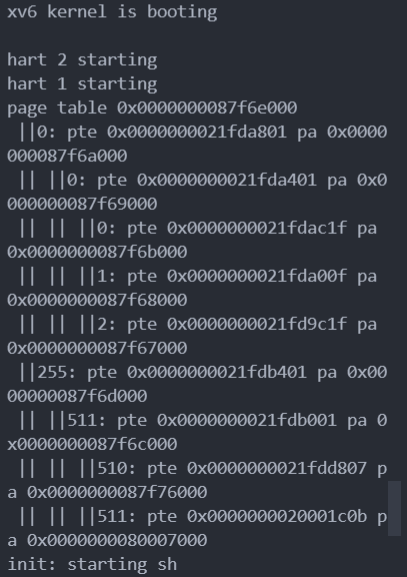
{

  printf("page table %p\n", pagetable);

  vmprint\_aux(pagetable, 0);

}

4、测试



黑盒测试通过

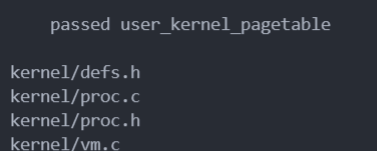
# 任务二：独立内核页表

1、任务

现在，xv6的每个进程都有自己独立的 用户页表 ，但是每个进程进入内核的时候，会使用唯一的一个 共享内核页表 。我们需要 将共享内核页表改成独立内核页表 ，使得每个进程拥有自己独立的内核页表。并处理好：分配、回收、页表切换等处的代码逻辑。

2、思路

需要修改的部分涉及到以下方面：



1. 在vm.c中增加ukvmcreate

因为kvminit()是初始化pagetable\_t kernel\_pagetable;全局变量，所以我们需要设计一个函数ukvmcreate来创建任意的kernel\_pagetable，注意函数体内应当避免CLINT段的重复初始化。

虽然在ukvmcreate中可以直接复用mappages通用函数，但是模仿kvmmap的写法我们写了一个kvmmap\_包装，输出panic情况。

1. 在proc.c中修改allocproc

扩展allocproc函数中found的逻辑，在此处初始化ukvm（init user's kernel pagetable），然后将原本存储在kernel\_pagetable中的p->kstack转移到ukvm中来。这里有一个细节，就是需要预先重声明kernel\_pagetable 这个外部变量，这样变量kernel\_pagetable 才可用，语句是extern pagetable\_t kernel\_pagetable; // from vm.c。

1. 在proc.c中修改scheduler中的调度逻辑

主要是需要迁移，原本隐式使用kernel\_pagetable记录的satp寄存器来定位进程上下文的地方，现在应当使用进程的p->pagetable\_k记录的satp寄存器来定位。

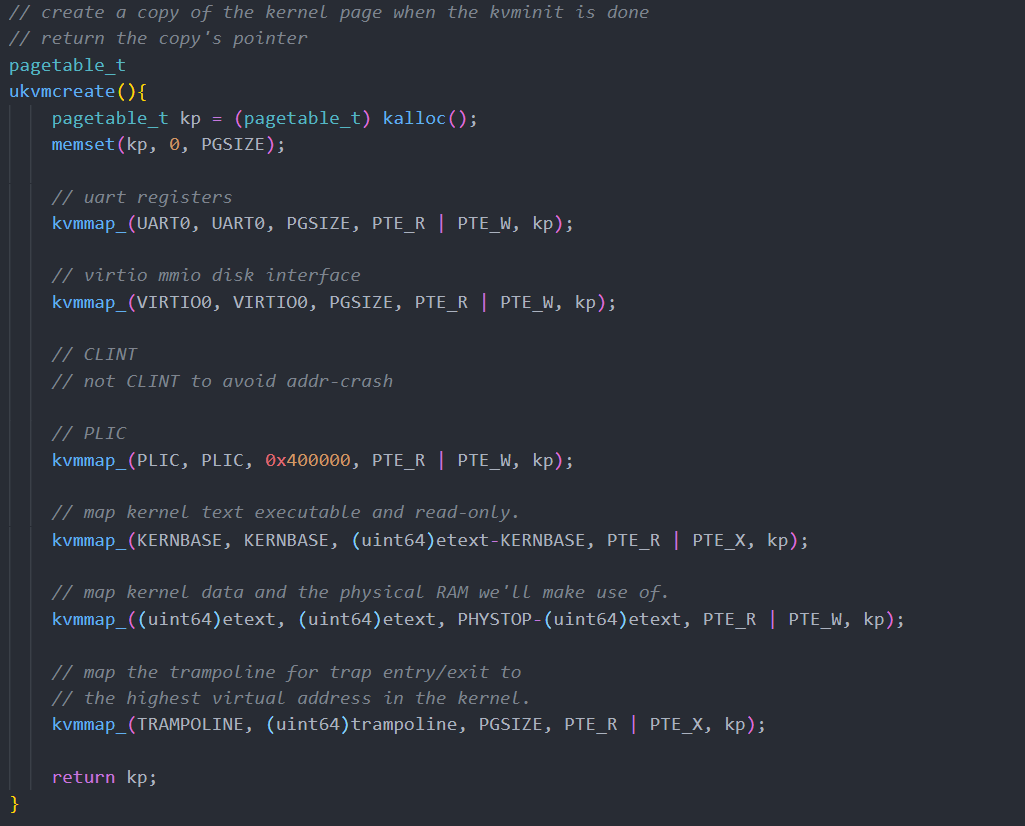
1. 在proc.c中修改freeproc的析构逻辑

增加对ukvm的析构部分。

3、代码

1. 在vm.c中增加ukvmcreate

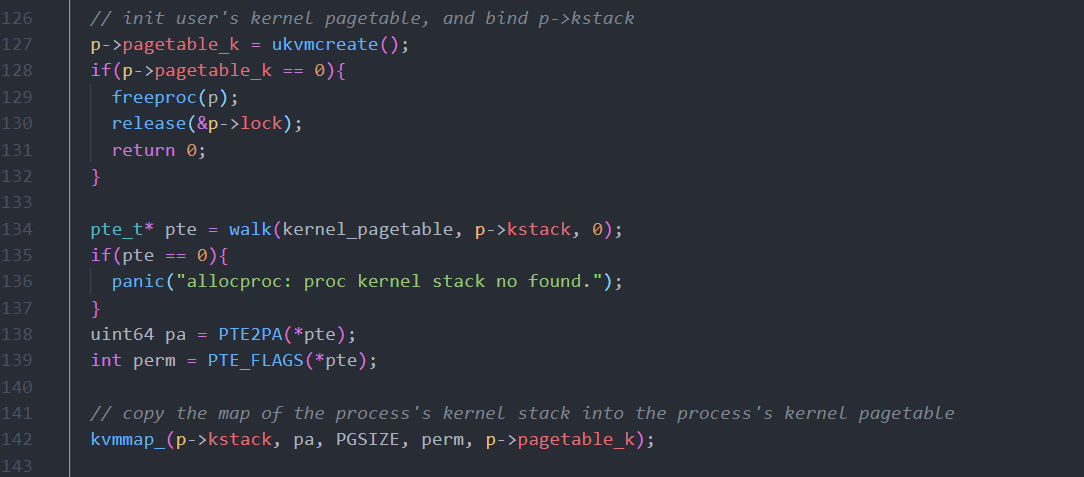
直接copy kvminit函数的逻辑



注意避开CLINT段。

2. 在proc.c中修改allocproc

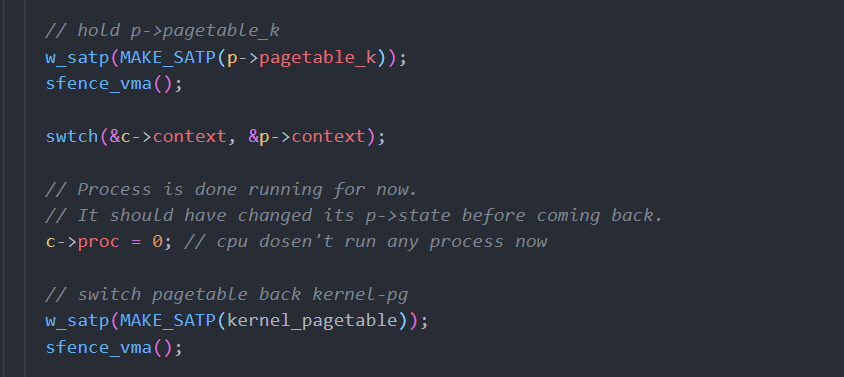
增加的逻辑如下



主要是增加了pagetable\_k结构体的实例化，和p->kstack pte的复制。注意，这里调用了walk函数，需要在defs.h文件中注册，缺省是未注册的估计原先只想暴露walkaddr函数，但是现在查找pte仍然需要使用walk函数。

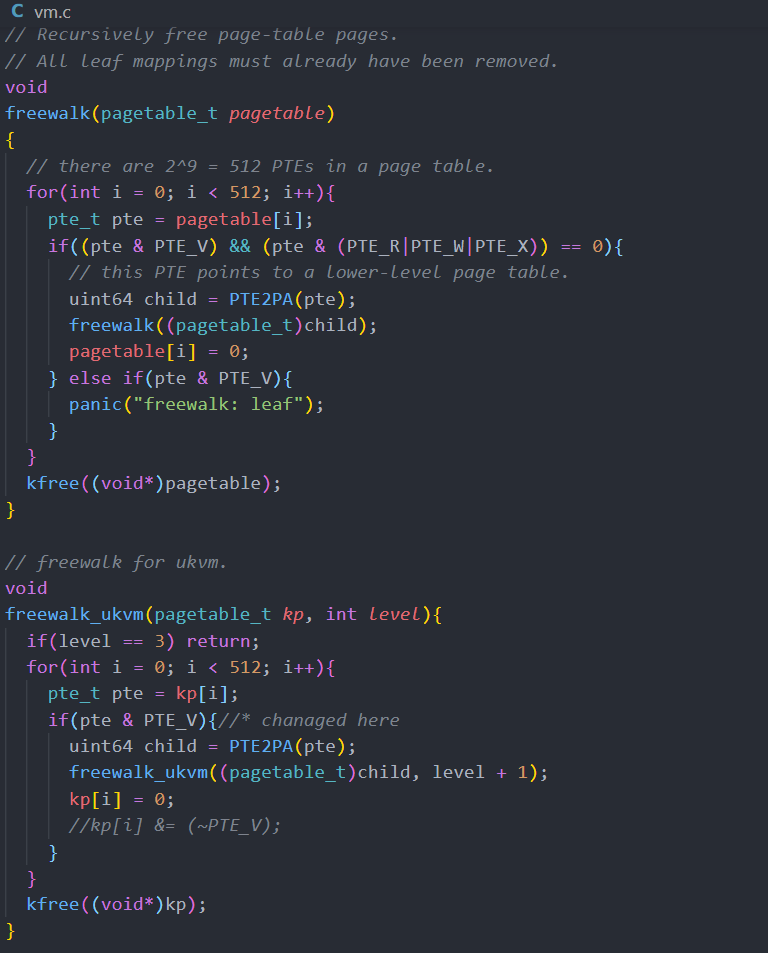
3. 在proc.c中修改scheduler中的调度逻辑

在swtch函数的两侧注意使用w\_stap函数以及sfence\_vma函数进行包裹，至于原因参考w\_stap函数以及sfence\_vma函数的定义。



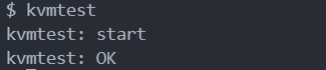
4. 在proc.c中修改freeproc的析构逻辑

在这里仍然是采取递归的方式，但是不能直接复用freewalk函数。可以对比一下两个函数（在vm.c下面）：



修改了过滤的条件，满足“释放页表但不释放叶子页表指向的物理页帧”这个条件。

4、测试



kvmtest黑盒测试通过



用户接口黑盒测试通过

# 任务三 简化软件模拟地址翻译

1. 任务

我们需要 在独立内核页表加上用户页表的映射，同时替换 copyin()/copyinstr() 为 copyin\_new()/copyinstr\_new() ，使得内核能够不必花费大量时间，用软件模拟的方法一步一步遍历页表，而是直接利用硬件，也就是复用上面任务二中设计的用户进程存储的内核页表。

1. 思路

下面约定，称用户页表为uvm，称用户进程存储的内核页表为ukvm，所以我们想要ukvm生效，首先要设计一个通用函数来**同步**uvm和ukvm，称为vmcopy。

该实验任务最难的部分就在于需要读源码查找修改uvm的位置，以进行同步。通过提示以及排查，发现需要修改的位置主要包括以下四个方面：fork，exec，growproc，userinit ，因为仅有这四个位置存在uvm相关的逻辑，所以我们需要在这些位置补充对ukvm的支持，通过通用的vmcopy函数。

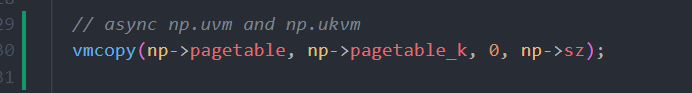
既然背后对ukvm的支持完善了，我们就可以大胆地进行copyin\_new和copyinstr\_new的实现。

1. 代码
2. 通用函数vmcopy的设计：
3. *// async uvm and ukvm*
4. *// impl: copy page from uvm pgtbl to ukvm pgtbl*
5. *// use case:*
6. *//   newsz=0 => del up*
7. *//   odlsz=0 => new kp*
8. int
9. vmcopy(pagetable\_t *up*, pagetable\_t *kp*, uint64 *oldsz*, uint64 *newsz*)
10. {
11. uint64 i,pa;
12. uint flags;
13. pte\_t \*pte=0;
14. if(oldsz < newsz)
15. {
16. oldsz = PGROUNDUP(oldsz);
17. for(i=oldsz; i<newsz; i+=PGSIZE)
18. {
19. if((pte=walk(up, i, 0)) == 0)
20. panic("vmcopy: extended pte no found in uvm.");
22. if((\*pte & PTE\_V) == 0)
23. panic("vmcopy: extended pte not present.");
25. pa = PTE2PA(\*pte);
26. flags = PTE\_FLAGS(\*pte);
27. *//clear the user flag when converting to kvm page.*
28. flags &= (~PTE\_U);
30. if((pte=walk(kp, i, 1)) == 0)
31. panic("vmcopy: extended pte no found in kvm.");
32. \*pte = PA2PTE(pa) | flags;*//concat*
33. }
34. }  else
35. {
36. newsz = PGROUNDUP(newsz);
37. for(i=newsz; i<oldsz; i+=PGSIZE)
38. {
39. if((pte=walk(kp, i, 0))  == 0)
40. panic("vmcopy: extended pte no found in kvm.");
41. if((\*pte & PTE\_V) == 0)
42. panic("vmcopy: extended pte not present.");
44. *//delete pte*
45. \*pte = 0;
46. }
47. }
48. return 0;
49. }

vmcopy既然是同步ukvm和uvm，本质就是不对称的页进行复制，逻辑类似于uvmcopy函数，后者是将父进程的uvm复制到子进程的uvm中。

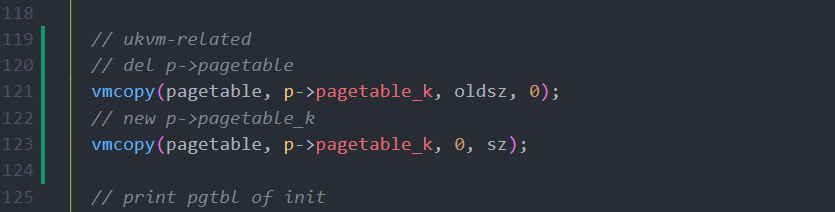
vmcopy的用例，典型的就是oldsz和newsz都不为0，表示同步uvm到ukvm。非典型的用例包括：oldsz=0，表示新建ukvm；newsz=0，表示删除原有的uvm。这两个用法都会出现。

2. 在fork中增加新建子进程的ukvm的逻辑：



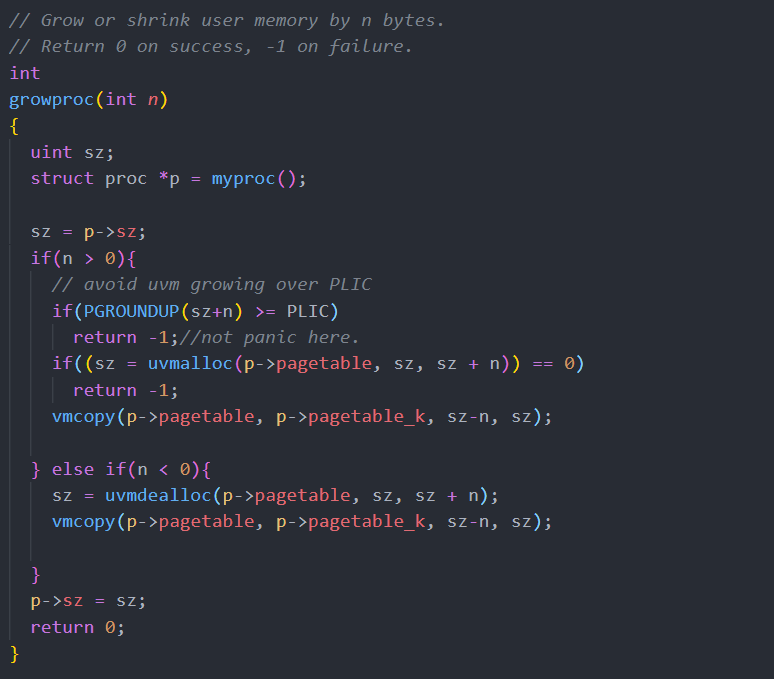
很直接的新建子进程的ukvm。

3.在exec中增加先删除ukvm后新建ukvm的逻辑：



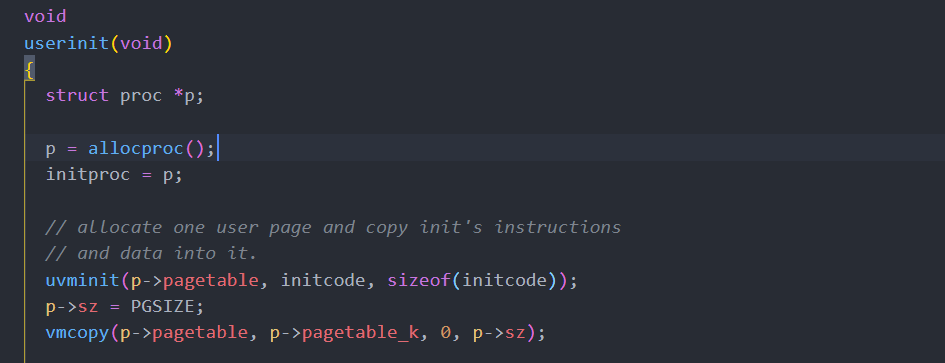
这里值得注意的是：需要先调用newsz=0的用例来清除pagetable\_k，这取决于fork的COW的特性。当fork+exec组合使用的时候，在exec中必须先清除此前存在的pagetable\_k结构。

4. 在growproc中增加同步uvm和ukvm的逻辑：



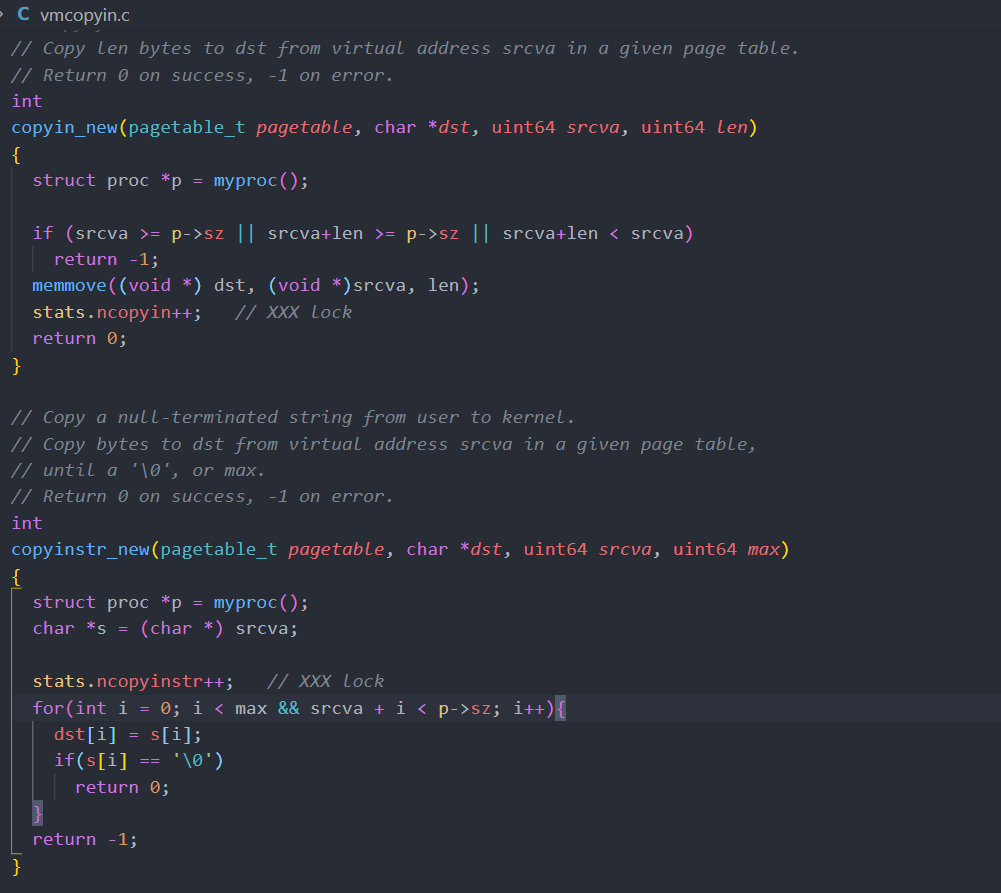
growproc函数的作用是调整proc的uvm的大小，所以在if n>0分支也就是增加uvm的情况下，需要进行if(PGROUNDUP(sz+n) >= PLIC)的判断，来避免越过PLIC的情况。

5.在userinit中增加新建ukvm的逻辑：



和fork的用例类似，不赘述。

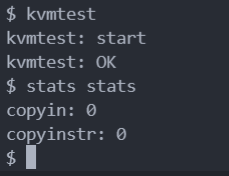
6.然后注意到已经提供的copyin\_new和copyinstr\_new的实现：



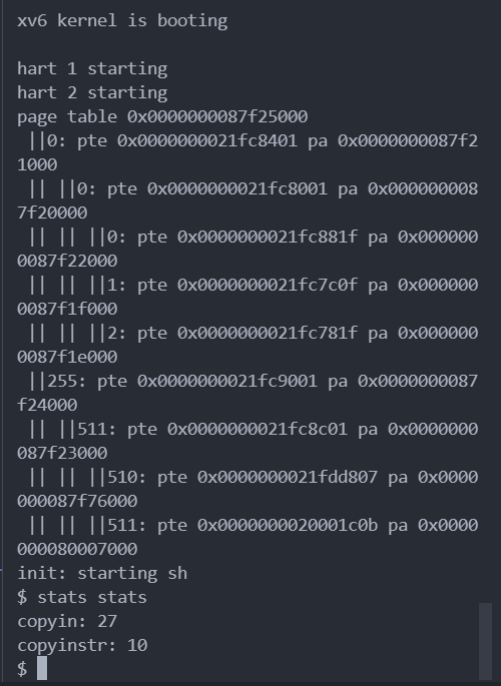
这个是不能直接移植到vm.c文件中的，因为缺少环境比如stats变量。

1. 测试

修改之前的stats测试确实为0



现在确实不为0了，说明我们对copyin\_new和copyinstr\_new的复用是成功的。



usertests黑盒测试：通过

