# Lab3 page tables

## Print a page table

### 实验目的：

Define a function called vmprint(). It should take a pagetable\_t argument, and print that pagetable in the format described below. Insert if(p->pid==1) vmprint(p->pagetable) in exec.c just before the return argc, to print the first process’s page table. You receive full credit for this assignment if you pass the pte printout test of make grade.

设计一个vmprint函数，打印页表的内容，运用到了广度优先搜索BFS

### 实验步骤：

1. 在exec.c中插入

  if(p->pid==1) vmprint(p->pagetable);

使得该函数在初始时能被执行

1. 在exec.c中声明vmprint()函数

void vmprint(pagetable\_t pagetable;

1. 在vm.c中参考freewalk函数递归实现vmprint函数

void printwalk(pagetable\_t pagetable, int depth)

{

  // there are 2^9 = 512 PTEs in a page table.

  for(int i = 0; i < 512; i++){

    pte\_t pte = pagetable[i];

    if(pte & PTE\_V){

      for (int j = 0;j < depth;j++){

        printf("..");

        if (j != depth - 1)

          printf(" ");

      }

      // this PTE points to a lower-level page table.

      // type cast

      uint64 child = PTE2PA(pte);

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

      if ((pte & (PTE\_R|PTE\_W|PTE\_X)) == 0){

        printwalk((pagetable\_t)child, depth + 1);

      }

    }

  }

}

void vmprint(pagetable\_t pagetable){

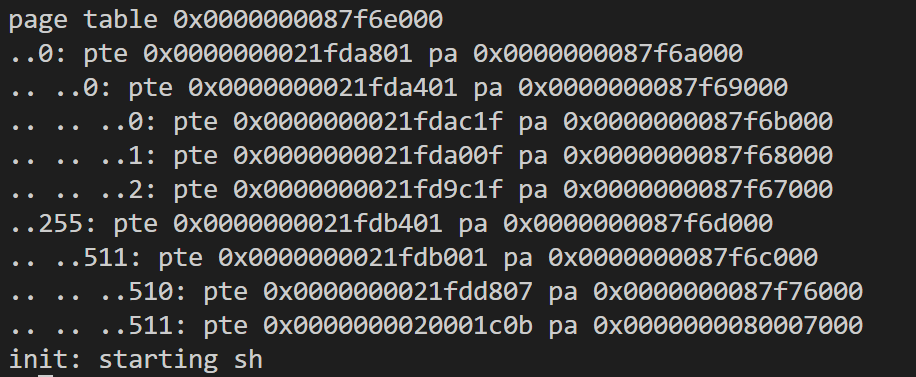
  printf("page table %p\n", pagetable);

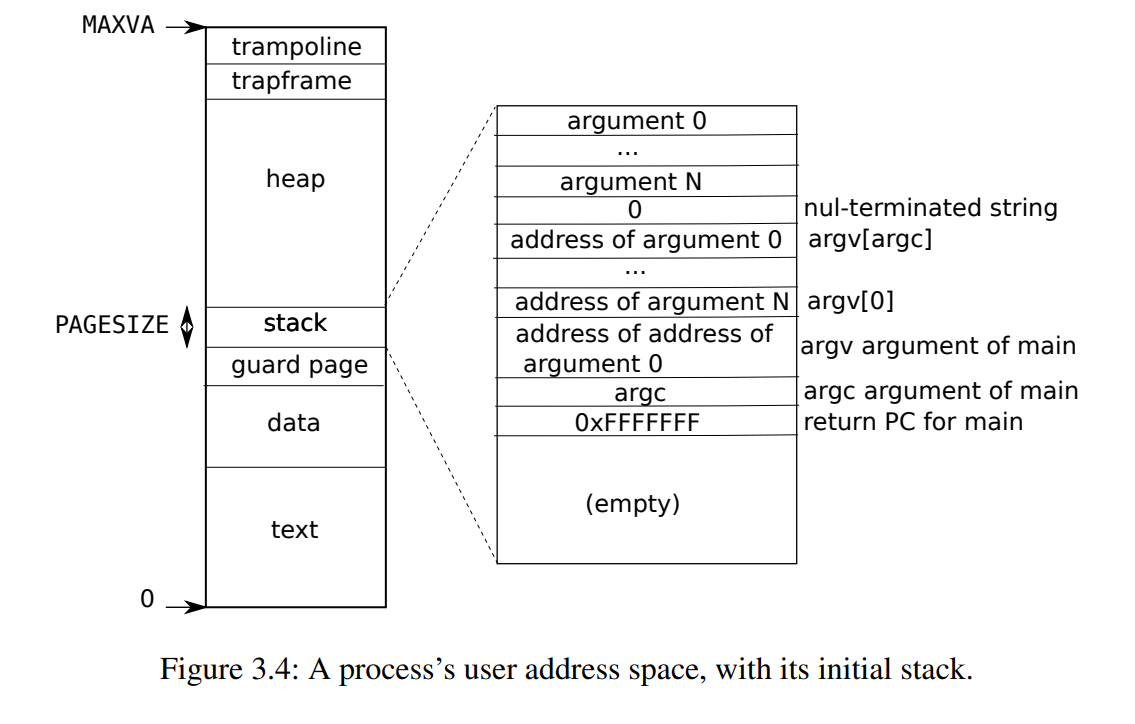
  printwalk(pagetable, 1);

}

PTE\_\*位表示riscv的地址的某一特定位，如PTE\_W就表示是否允许写，PTE\_V表示是否有效

打印结果如下：





### 实验中遇到的问题及解决办法:

查阅相关文献和上图可知，page0对应程序的代码段和数据段，page2则对应用户栈，中间的page1是guard page不能用于映射，所以当运行在用户模式时，进程不可以读/写第1页映射的内存。

### 实验心得：

本实验主要还是理解好三级页表的逻辑存储方式，9 bit 一级索引找到二级页表，9 bit 二级索引找到三级页表，9 bit 三级索引找到内存页，最低 12 bit 为页内偏移（即一个页 4096 bytes），最后通过函数将之打印出来就好了。

## A kernel page table per process

### 实验目的：

Your first job is to modify the kernel so that every process uses its own copy of the kernel page table when executing in the kernel. Modify struct proc to maintain a kernel page table for each process, and modify the scheduler to switch kernel page tables when switching processes. For this step, each per-process kernel page table should be identical to the existing global kernel page table. You pass this part of the lab if usertests runs correctly.

这个实验允许内核直接解除对用户指针的引用，也就是给每个进程配备一个内核页表。xv6原本只有一个内核页表，当处理user进程传入的指针时，只能先处理为物理地址（physical address）再处理。

1. 在proc.h中添加用于进程内核页表的字段

pagetable\_t kernelpgtbl;     // add kernel page

1. 修改kvminit()函数。

该函数原本的逻辑是将全局变量kernel\_pagetable，即xv6原本唯一的内核页表初始化并map。而修改后需要先在另一个函数kvm\_init\_one里初始化一个内核页表，再在kvminit里调用此函数，复制给内核页表。

在kernel/vm.c下实现以下函数：

pagetable\_t kvm\_init\_one(){

  pagetable\_t newpg = uvmcreate();

  // copy but forget to modify the first argument. so sad

  // uart registers

  kvmmap\_with\_certain\_page(newpg, UART0, UART0, PGSIZE, PTE\_R | PTE\_W);

  // virtio mmio disk interface

  kvmmap\_with\_certain\_page(newpg, VIRTIO0, VIRTIO0, PGSIZE, PTE\_R | PTE\_W);

  // CLINT

  kvmmap\_with\_certain\_page(newpg, CLINT, CLINT, 0x10000, PTE\_R | PTE\_W);

  // PLIC

  kvmmap\_with\_certain\_page(newpg, PLIC, PLIC, 0x400000, PTE\_R | PTE\_W);

  // map kernel text executable and read-only.

  kvmmap\_with\_certain\_page(newpg, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE\_R | PTE\_X);

  // map kernel data and the physical RAM we'll make use of.

  kvmmap\_with\_certain\_page(newpg, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE\_R | PTE\_W);

  // map the trampoline for trap entry/exit to

  // the highest virtual address in the kernel.

  kvmmap\_with\_certain\_page(newpg, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE\_R | PTE\_X);

  return newpg;

}

/\*

 \* create a direct-map page table for the kernel.

 \*/

void

kvminit()

{

  kernel\_pagetable = kvm\_init\_one();

}

// simulate kvmmap

void kvmmap\_with\_certain\_page(pagetable\_t pg, uint64 va, uint64 pa, uint64 sz, int perm)

{

  if(mappages(pg, va, sz, pa, perm) != 0)

    panic("kvmmap with certain page");

}

在def.h中添加声明：

void kvmmap\_with\_certain\_page(pagetable\_t pg, uint64 va, uint64 pa, uint64 sz, int perm);

void kvmmap\_with\_certain\_page(pagetable\_t pg, uint64 va, uint64 pa, uint64 sz, int perm);

pagetable\_t kvm\_init\_one();

pte\_t \* walk(pagetable\_t pagetable, uint64 va, int alloc);

除此之外，不要忘记在vm.c中添加以下头文件

#include "spinlock.h"

#include "proc.h"

3.在原先xv6中，所有内核栈均设置在`procinit`函数中初始化，为实现本实验功能，需将初始化移动到`allocproc`中，并调用刚才写的函数。

在proc.c中将procinit中的以下代码注释

      // char \*pa = kalloc();

      // if(pa == 0)

      //   panic("kalloc");

      // uint64 va = KSTACK((int) (p - proc));

      // kvmmap(va, (uint64)pa, PGSIZE, PTE\_R | PTE\_W);

      // p->kstack = va;

在allocproc中添加以下代码：

  // add kernel page table

  p->kernelpgtbl = kvm\_init\_one();

  if (p->kernelpgtbl == 0){

    freeproc(p);

    release(&p->lock);

    return 0;

  }

  if(pa == 0)

    panic("kalloc");

  uint64 va = KSTACK((int) (p - proc));

  kvmmap\_with\_certain\_page(p->kernelpgtbl, va, (uint64)pa, PGSIZE, PTE\_R | PTE\_W);

  p->kstack = va;

4.接下来修改scheduler，使其在切换进程的同时切换内核页表（记得刷新TLB！），在该进程结束后（and没有运行进程的时候）记得换回kernel\_pagetable

        p->state = RUNNING;

        c->proc = p;

        // Switch to the independent kernel page table

        w\_satp(MAKE\_SATP(p->kernelpgtbl));

        // flush TLB

        sfence\_vma();

        swtch(&c->context, &p->context);

        // Switch back to global kernel page table

        kvminithart();

        // Process is done running for now.

        // It should have changed its p->state before coming back.

        c->proc = 0;

5.修改freeproc函数，使其能正确释放内核页表和内核栈

  if (p->kstack){

    pte\_t\* pte = walk(p->kernelpgtbl, p->kstack, 0);

    if (pte == 0)

      panic("freeproc: kstack");

    kfree((void\*)PTE2PA(\*pte));

  }

  p->kstack = 0;

  if(p->pagetable)

    proc\_freepagetable(p->pagetable, p->sz);

  p->pagetable = 0;

  if (p->kernelpgtbl){

    kvm\_free\_pgtbl(p->kernelpgtbl);

  }

  p->kernelpgtbl = 0;

// free pg recursively

void kvm\_free\_pgtbl(pagetable\_t pg){

  for (int i = 0; i < 512; i++){

    pte\_t pte = pg[i];

    // copy wrong!!

    // if((pte & PTE\_V) && (PTE\_R|PTE\_W|PTE\_X) == 0){

    if((pte & PTE\_V) && (pte & (PTE\_R|PTE\_W|PTE\_X)) == 0){

      uint64 child = PTE2PA(pte);

      kvm\_free\_pgtbl((pagetable\_t)child);

      pg[i] = 0;

    }

  }

  kfree((void\*)pg);

}

1. 最后修改kvmpa，将默转换的内核页表替换为正在运行的进程

  pte\_t \*pte;

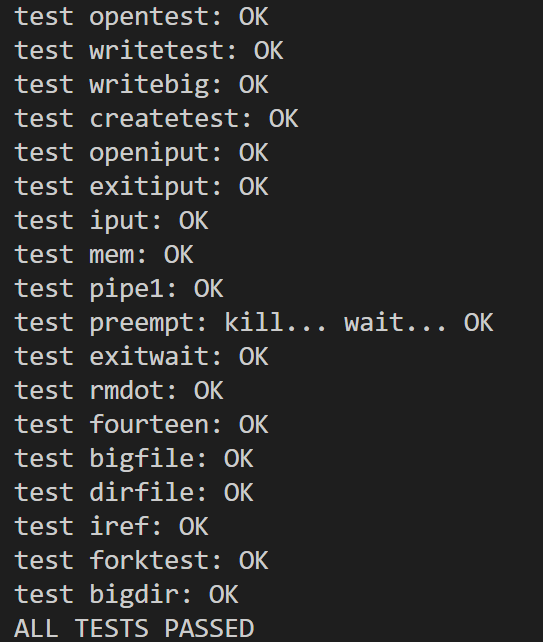
  uint64 pa;

  pte = walk(myproc()->kernelpgtbl, va, 0);

  if(pte == 0)

    panic("kvmpa");

至此，内核页表的修改流程结束，在终端运行usertests，可以得到以下正确结果：



### 实验中遇到的问题和解决办法：

单元测试失败，报错`/usr/bin/env: ‘python’: No such file or directory`

解决办法：进入/user/bin/env目录，发现没有’pyhon’而是’python3’。之后进入实验原码目录下的’./grade-lab-traps’，第一行中的’python’修改为’python3’即可

如果执行usertests在测试reparent2的时候出现了 panic: kvmmap，大概率是因为大量内存泄漏消耗完了内存，导致 kvmmap 分配页表项所需内存失败，这时候应该检查是否正确释放了每一处分配的内存，尤其是页表是否每个页表项都释放干净了。

实验心得：

这次实验我解决问题的方式主要是‘模仿’，例如模仿freewalk、实现一些系统调用，在实现一些比较复杂的函数时，我们可以先从类似的函数入手，先仿照这些函数如实现类似的功能，然后再针对该函数自己特定的功能来继续修改参数、逻辑。

## Simplify copyin/copyinstr

### 实验目的：

Replace the body of copyin in kernel/vm.c with a call to copyin\_new (defined in kernel/vmcopyin.c); do the same for copyinstr and copyinstr\_new. Add mappings for user addresses to each process’s kernel page table so that copyin\_new and copyinstr\_new work. You pass this assignment if usertests runs correctly and all the make grade tests pass.

用`copyin[str]\_new`代替原先的`copyin[str]`函数，同时修改user mapping，使用户地址能map到每个进程的内核页表中

### 实验步骤：

1.先在kernel/defs.h中声明

// copy user pagetable to kernel page table

void            uvm2kvm(pagetable\_t u, pagetable\_t k, uint64 from, uint64 to);

// copy new

int             copyin\_new(pagetable\_t pagetable, char \*dst, uint64 srcva, uint64 len);

int             copyinstr\_new(pagetable\_t pagetable, char \*dst, uint64 srcva, uint64 max);

```

2.再在kernel/vm.c中实现`uvm2kvm`函数，将用户页表转换到内核页表。注意PLIC限制，同时将PTE\_U设为0。

void uvm2kvm(pagetable\_t userpgtbl, pagetable\_t kernelpgtbl, uint64 from, uint64 to)

{

  if (from > PLIC) // PLIC limit

    panic("uvm2kvm: from larger than PLIC");

  from = PGROUNDDOWN(from); // align

  for (uint64 i = from; i < to; i += PGSIZE)

  {

    pte\_t \*pte\_user = walk(userpgtbl, i, 0);

    pte\_t \*pte\_kernel = walk(kernelpgtbl, i, 1);

    if (pte\_kernel == 0)

      panic("uvm2kvm: allocating kernel pagetable fails");

    \*pte\_kernel = \*pte\_user;

    \*pte\_kernel &= ~PTE\_U;

  }

}

3.某些系统调用（fork(), exec(), sbrk()）会改变user mapping，需要对此也做相应更改。

fork：

  np->sz = p->sz;

  uvm2kvm(np->pagetable, np->kernelpgtbl, 0, np->sz);

  np->parent = p;

exec：

  // add vmprint

  if(p->pid==1) vmprint(p->pagetable);

  uvm2kvm(p->pagetable, p->kernelpgtbl, 0, p->sz);

  return argc; // this ends up in a0, the first argument to main(argc, argv)

userinit：

  uvminit(p->pagetable, initcode, sizeof(initcode));

  p->sz = PGSIZE;

  uvm2kvm(p->pagetable, p->kernelpgtbl, 0, p->sz); // copy from user to kernel

  // prepare for the very first "return" from kernel to user.

  p->trapframe->epc = 0;      // user program counter

growproc：

  } else if(n < 0){

    sz = uvmdealloc(p->pagetable, sz, sz + n);

  }

  uvm2kvm(p->pagetable, p->kernelpgtbl, sz - n, sz);

  p->sz = sz;

  return 0;

}

最后替换copyin[str]：

int

copyinstr(pagetable\_t pagetable, char \*dst, uint64 srcva, uint64 max)

{

  return copyinstr\_new(pagetable, dst, srcva, max);

}

int

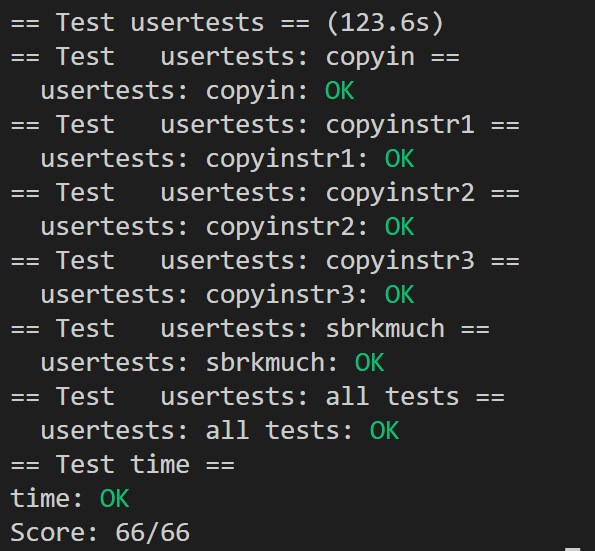
copyin(pagetable\_t pagetable, char \*dst, uint64 srcva, uint64 len)

{

  return copyin\_new(pagetable, dst, srcva, len);

}

实验结果如下：



### 实验中遇到的问题和解决办法：

解释为什么`srcva + len < srcva`是必要的：例如，srcva=0x10，len为0xffff...ffff时，满足srcva >= p->sz，srcva+len >= p->sz，但srcva+len溢出，小于srcva，便可以检测到溢出

实验指导中说At each point where the kernel changes a process's user mappings, change the process's kernel page table in the same way的意思不是为每一处用户页表被映射的地方为内核页表施用同样或类似的函数，而是仅需将内核页表的虚拟地址映射至用户页表虚拟地址所映射的物理地址即可，不需要为内核页表分配新的物理内存，这里会产生一些误解。

### 实验感想：

做实验之前还是得看实验中提及的文献和视频，在不理解概念的情况下做实验实在太艰难了，即使跟着教程做完了实验也仍是处于一知半解的状态，还是得把最本质的知识弄明白。