Lab3：Page tables

该实验由Print a page table**、**Detecting which pages have been accessed两部分组成。

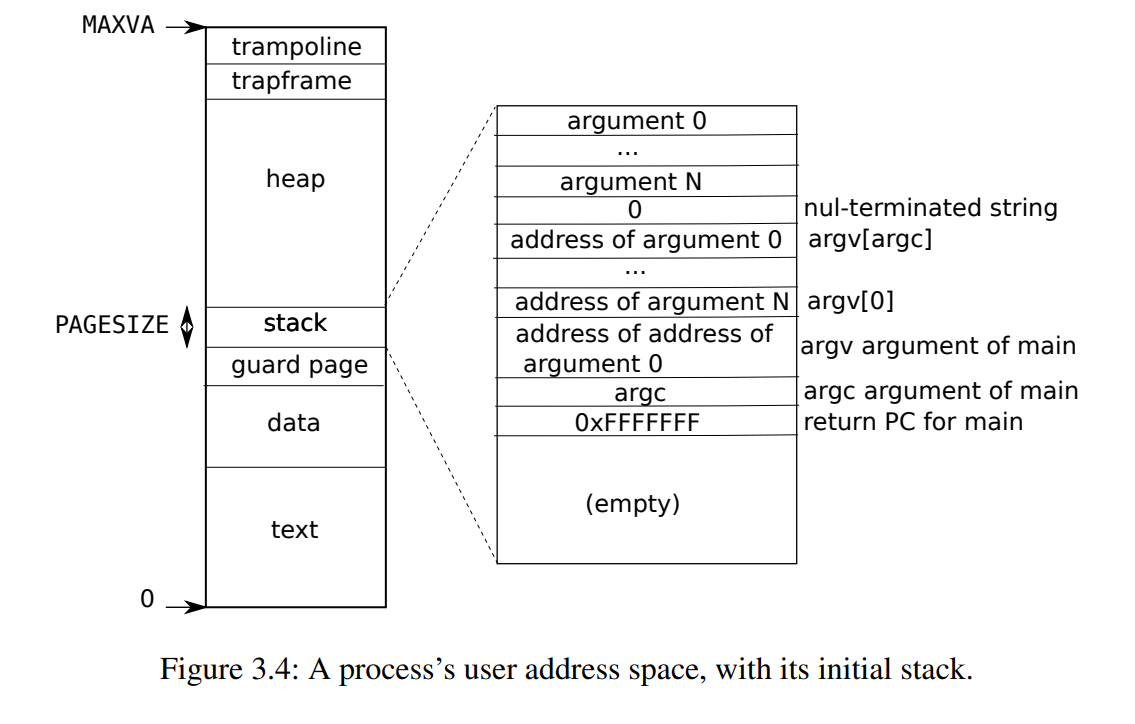
1. Print a page table

实验目的

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

实验原理如下：



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){

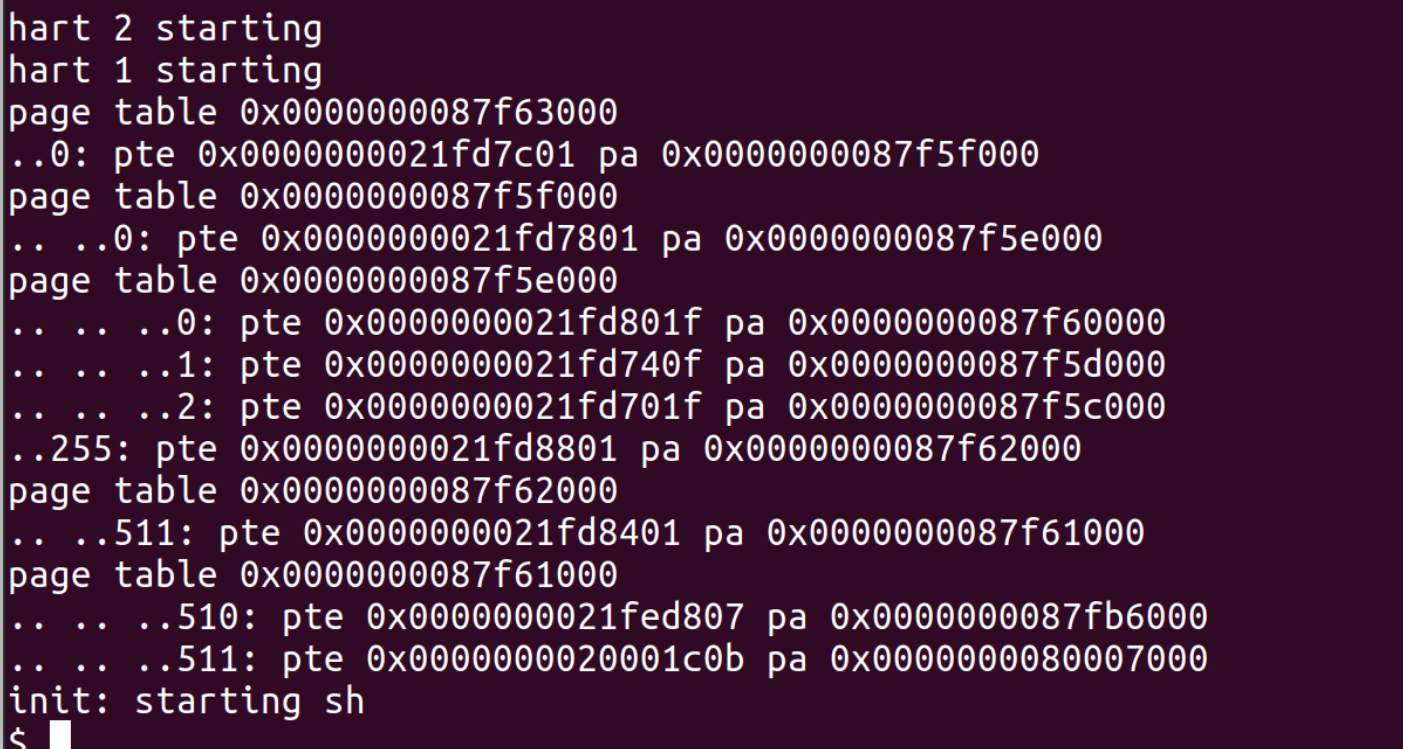
  printf("page table %p\n", pagetable);

  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：



遇到的问题与心得

遇到的问题：

格式化输出问题。

解决办法：结合控制台调试，使用 %p 格式化符正确地打印了 64 位的十六进制 PTE 和物理地址。

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。

2. Detecting which pages have been accessed

实验目的

本实验的目的是向 xv6 内核添加一个新特性，即通过检查 RISC-V 页表中的访问位，实现一个系统调用 pgaccess()，该系统调用可以报告哪些页面已被访问（读取或写入）。这对于垃圾回收器等应用有一定的益处。

实验步骤

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;

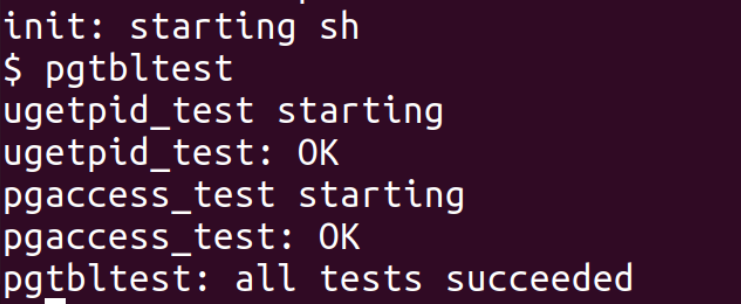
  uint64 pa;

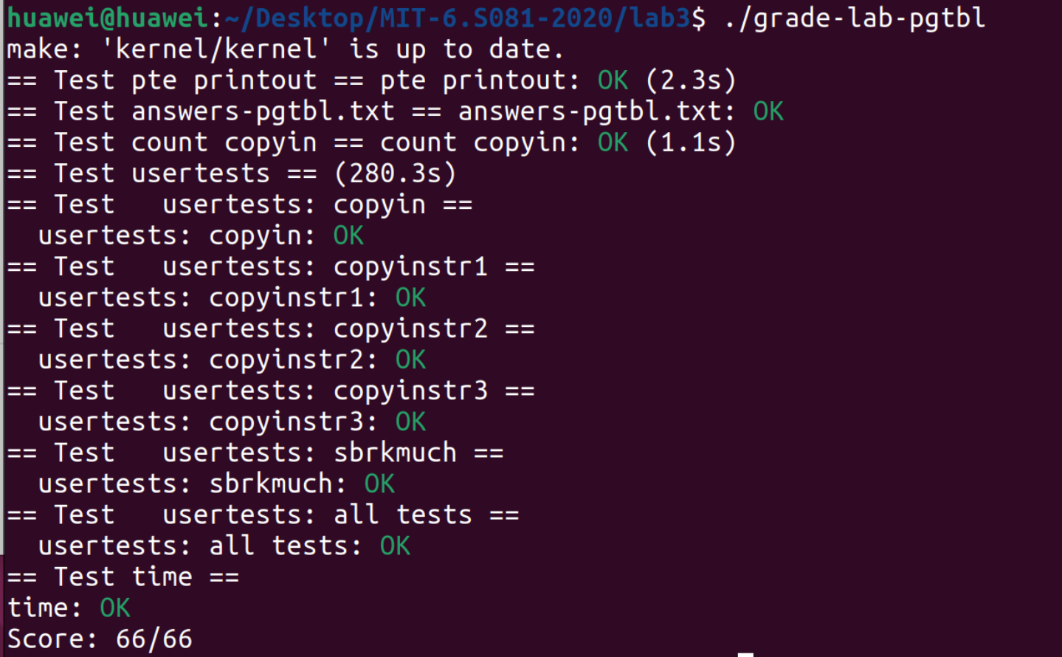
  pte = walk(myproc()->kernelpgtbl, va, 0);

  if(pte == 0)

panic("kvmpa");

7.保存后在终端里执行make qemu编译运行xv6：





遇到的问题与心得

遇到的问题：

在实现 pgaccess() 系统调用功能时，未正确遍历用户页并检查 PTE\_A 位。

解决办法：可以通过公式公式：

`x = ((x&(1 << n)) ^ x) ^ (a << n)`。

x 为原值，n 为第 n 个值，a 为想要设置的值（0或1）。

首先 (x&(1 << n)) 的值为：保留第 n 位原来的值，其他位置零。再将此值与原值 x 异或，得到一个值：除了第 n 个值为零，其他位置为原值。（这是因为，与 0 异或的那一位为原值，与相同值异或的那一位为 0）。然后此时，再与 (a << n) 异或，将第 n 位设置为 a （这是因为与 0 异或为其本身）。

实验心得：

通过这个实验，我学习了操作系统的内存管理机制，包括页表的结构和作用；理解了如何为进程分配页表，映射虚拟地址到物理地址，以及如何使用页表权限来实现不同的访问控制。

在实现系统调用过程中，我深入了解了内核代码的组织结构和运行方式，以及如何将用户态的请求转换为内核态的操作，并了解了如何从用户空间传递参数到内核空间。

此外我还能够结合教材中的页表结构来编写程序，这更让我理解了上层算法实现和底层硬件逻辑之间的紧密联系，这为我今后的学习打下基础、积累了一定的经验。