**实验三 page tables**

**3.1 实验目的**

1. 了解页表机制及其结构

2. 了解内核虚拟地址和物理地址的映射关系

3. 通过实验自己尝试将用户映射添加到每个进程的内核页表上

**3.2 实验内容**

3.2.1 Print a page table（easy）

定义一个名为vmprint（）的函数。它应该采用pagetable\_t参数，并以下面描述的格式打印该pagetable。在返回argc之前的exec.c中插入if（p->pid==1）vmprint（p->pagetable），以打印第一个进程的页面表。如果您通过了pte printout的make grade，您将获得本次作业的全部学分。

当您启动xv6时，它应该像这样打印输出，描述第一个进程刚刚完成exec（）ing init时的页面表：

page table 0x0000000087f6e000

..0: pte 0x0000000021fda801 pa 0x0000000087f6a000

.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000

.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000

.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000

.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000

..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000

.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000

.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000

.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000

3.2.2 A kernel page table per process（hard）

您的第一项工作是修改内核，以便每个进程在内核中执行时都使用自己的内核页表副本。修改struct proc以维护每个进程的内核页表，并修改调度器以在切换进程时切换内核页表。对于这个步骤，每个进程的内核页表应该与现有的全局内核页表相同。如果usertests运行正常，则通过这部分实验。

3.2.3 Simplify copin/copyinstr（hard）

用对copyin\_new（在kernel/vmcopyin.c中定义）的调用替换kernel/vm.c中copyin的主体；对copyinstr和copyinstr\_new执行相同的操作。将用户地址映射添加到每个进程的内核页表中，以便copyin\_new和copyinstr\_new工作。如果usertests正确运行并且所有make grade测试都通过，则通过此实验。

3.2.4 submit

提交实验。

**3.3 实验过程**

3.3.1 print a page table

（1）为实现vmprint()函数我们需要实现vmprintdfs()来递归调用页表中的每一项并输出，因为xv6采用了三级页表机制，因此只需要递归到第三层。

void

vmprintdfs(pagetable\_t pt,int deep)

{

  for(int i=0;i<512;i++){

    pte\_t pte=pt[i];

    if(pte & PTE\_V){

      for(int i=0;i<deep;i++)

        printf(".. ");

      uint64 child=PTE2PA(pte);

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

      if(deep==3)

        continue;

      else

        vmprintdfs((pagetable\_t)child,deep+1);

    }

  }

}

（2）编写vmprint()函数调用vmprintdfs()并进行输出。

void

vmprint(pagetable\_t pt)

{

  printf("page table %p\n",pt);

  vmprintdfs(pt,1);

}

（3）为运行vmprint()函数，我们需要在defs.h中对该函数进行声明并在exec.c中添加对于vmprint()的调用

defs.h

void            vmprint(pagetable\_t pt);

exec.c

  if(p->pid==1)

    vmprint(p->pagetable);

（4）运行vmprint



3.3.2 a kernel page table per process

（1）每个进程都有一个单独的页表，用于每个进程的 用户地址空间映射，只包含该进程的用户内存的映射，当内核需要使用系统调用中传递的用户指针时，需要先将指针翻译成物理地址。所以我们需要在进程proc结构体中增加一个内核页表。（proc.h）

// Per-process state

struct proc {

  struct spinlock lock;

  // p->lock must be held when using these:

  enum procstate state;        // Process state

  struct proc \*parent;         // Parent process

  void \*chan;                  // If non-zero, sleeping on chan

  int killed;                  // If non-zero, have been killed

  int xstate;                  // Exit status to be returned to parent's wait

  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.

  uint64 kstack;               // Virtual address of kernel stack

  uint64 sz;                   // Size of process memory (bytes)

  pagetable\_t pagetable;       // User page table

  struct trapframe \*trapframe; // data page for trampoline.S

  struct context context;      // swtch() here to run process

  struct file \*ofile[NOFILE];  // Open files

  struct inode \*cwd;           // Current directory

  char name[16];               // Process name (debugging)

  pagetable\_t kpagetable;

};

（2）在vm.c中模仿kvminit()增加kvmmake()方法用来创建内核页表，同时将kvmmake()方法加入defs.h中。

pagetable\_t kvmmake()

{

  pagetable\_t kpagetable = (pagetable\_t)kalloc();

  memset(kpagetable, 0, PGSIZE);

  // uart registers

  kvmmap(kpagetable, UART0, UART0, PGSIZE, PTE\_R | PTE\_W);

  // virtio mmio disk interface

  kvmmap(kpagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE\_R | PTE\_W);

  // CLINT

  kvmmap(kpagetable, CLINT, CLINT, 0x10000, PTE\_R | PTE\_W);

  // PLIC

  kvmmap(kpagetable, PLIC, PLIC, 0x400000, PTE\_R | PTE\_W);

  // map kernel text executable and read-only.

  kvmmap(kpagetable, KERNBASE, KERNBASE, (uint64)etext - KERNBASE, PTE\_R | PTE\_X);

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

  kvmmap(kpagetable, (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(kpagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE\_R | PTE\_X);

  return kpagetable;

}

（3）因为对kvmmap()函数中参数进行了修改，所以修改kvmmap()函数并相应的在defs.h中进行更改

void kvmmap(pagetable\_t kpagetable, uint64 va, uint64 pa, uint64 sz, int perm)

{

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

    panic("kvmmap");

}

// translate a kernel virtual address to

// a physical address. only needed for

// addresses on the stack.

// assumes va is page aligned.

uint64

kvmpa(uint64 va)

{

  uint64 off = va % PGSIZE;

  pte\_t \*pte;

  uint64 pa;

  // pte = walk(kernel\_pagetable, va, 0);

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

  if (pte == 0)

    panic("kvmpa");

  if ((\*pte & PTE\_V) == 0)

    panic("kvmpa");

  pa = PTE2PA(\*pte);

  return pa + off;

}

（4）修改kvminit()进行kernel\_pagetable的初始化。

void kvminit()

{

  kernel\_pagetable = kvmmake();

}

（5）在proc.c中的allocproc()函数中完成kernel pagetable的初始化

static struct proc \*

allocproc(void)

{

  struct proc \*p;

  for (p = proc; p < &proc[NPROC]; p++)

  {

    acquire(&p->lock);

    if (p->state == UNUSED)

    {

      goto found;

    }

    else

    {

      release(&p->lock);

    }

  }

  return 0;

found:

  p->pid = allocpid();

  // Allocate a trapframe page.

  if ((p->trapframe = (struct trapframe \*)kalloc()) == 0)

  {

    release(&p->lock);

    return 0;

  }

  // A kernel page table

  if ((p->kpagetable = kvmmake()) == 0)

  {

    freeproc(p);

    release(&p->lock);

    return 0;

  }

  char \*pa = kalloc();

  if (pa == 0)

    panic("kalloc");

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

  kvmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE\_R | PTE\_W);

  p->kstack = va;

  // An empty user page table.

  p->pagetable = proc\_pagetable(p);

  if (p->pagetable == 0)

  {

    freeproc(p);

    release(&p->lock);

    return 0;

  }

  // Set up new context to start executing at forkret,

  // which returns to user space.

  memset(&p->context, 0, sizeof(p->context));

  p->context.ra = (uint64)forkret;

  p->context.sp = p->kstack + PGSIZE;

  return p;

}

（6）修改proc.c中的scheduler函数，在进程进行切换的时候把自己的kernel pagetable放入stap寄存器中，让内核使用该进程的pagetable转换地址，如果没有进程在运行就使用内核本身的kernel pagetable。

void

scheduler(void)

{

  struct proc \*p;

  struct cpu \*c = mycpu();

  c->proc = 0;

  for(;;){

    // Avoid deadlock by ensuring that devices can interrupt.

    intr\_on();

    int found = 0;

    for(p = proc; p < &proc[NPROC]; p++) {

      acquire(&p->lock);

      if(p->state == RUNNABLE) {

        // Switch to chosen process.  It is the process's job

        // to release its lock and then reacquire it

        // before jumping back to us.

        p->state = RUNNING;

        c->proc = p;

        // switch to porcess's kernel page table

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

        sfence\_vma();//flushes the current CPU's TLB

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

        // Process is done running for now.

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

        c->proc = 0;

        // switch to overall pagetable

        kvminithart();

        found = 1;

      }

      release(&p->lock);

    }

#if !defined (LAB\_FS)

    if(found == 0) {

      intr\_on();

      asm volatile("wfi");

    }

#else

    ;

#endif

  }

}

（7）在freeproc()中释放进程页表，不要释放物理内存。

static void

freeproc(struct proc \*p)

{

  if (p->trapframe)

    kfree((void \*)p->trapframe);

  p->trapframe = 0;

  if (p->kstack)

    uvmunmap(p->kpagetable, p->kstack, 1, 1);

  p->kstack = 0;

  if (p->kpagetable)

    kvmfree(p->kpagetable);

  p->kpagetable = 0;

  if (p->pagetable)

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

  p->pagetable = 0;

  p->sz = 0;

  p->pid = 0;

  p->parent = 0;

  p->name[0] = 0;

  p->chan = 0;

  p->killed = 0;

  p->xstate = 0;

  p->state = UNUSED;

}

其中kvmfree()为释放内核页表函数

void kvmfree(pagetable\_t kpgtbl)

{

  for (int i = 0; i < 512; i++)

  {

    pte\_t pte = kpgtbl[i];

    if (pte & PTE\_V)

    {

      kpgtbl[i] = 0;

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

      {

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

        uint64 child = PTE2PA(pte);

        kvmfree((pagetable\_t)child);

      }

    }

    else if (pte & PTE\_V)

    {

      panic("freewalk: leaf");

    }

  }

  kfree((void \*)kpgtbl);

}

（8）运行a kernel page table per process



3.3.3 simplify copyin/copyinstr

（1）首先修改vm.c中原本的copyin以及copyinstr函数，使其调用copyin\_new以及copyinstr\_new，并在defs.h中添加copyin\_new以及copyinstr\_new函数。

int

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

{

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

}

int

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

{

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

}

（2）实现uvm2k函数用于将进程的用户页表映射到内核页表。

int uvm2k(pagetable\_t pagetable, pagetable\_t kpgtbl, uint64 oldsz, uint64 newsz)

{

  pte\_t \*pte\_from, \*pte\_to;

  if ((newsz < oldsz) || (PGROUNDUP(newsz) >= PLIC))

    return -1;

  for (uint64 va = PGROUNDUP(oldsz); va < newsz; va += PGSIZE)

  {

    if ((pte\_from = walk(pagetable, va, 0)) == 0)

      panic("copyin\_new: pte not exist");

    if ((pte\_to = walk(kpgtbl, va, 1)) == 0)

      panic("copyin\_new: walk failed");

    \*pte\_to = (\*pte\_from) & (~PTE\_U);

  }

  return 0;

}

（3）根据提示对fork()，exec()函数进行修改 。

int fork(void)

{

  int i, pid;

  struct proc \*np;

  struct proc \*p = myproc();

  // Allocate process.

  if ((np = allocproc()) == 0)

  {

    return -1;

  }

  // Copy user memory from parent to child.

  if (uvmcopy(p->pagetable, np->pagetable, p->sz) < 0)

  {

    freeproc(np);

    release(&np->lock);

    return -1;

  }

  np->sz = p->sz;

  np->parent = p;

  // copy saved user registers.

  \*(np->trapframe) = \*(p->trapframe);

  // Cause fork to return 0 in the child.

  np->trapframe->a0 = 0;

  // increment reference counts on open file descriptors.

  for (i = 0; i < NOFILE; i++)

    if (p->ofile[i])

      np->ofile[i] = filedup(p->ofile[i]);

  np->cwd = idup(p->cwd);

  if (uvm2k(np->pagetable, np->kpagetable, 0, np->sz) < 0)

    return -1;

  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  np->state = RUNNABLE;

  release(&np->lock);

  return pid;

}

int exec(char \*path, char \*\*argv)

{

...

  sz = PGROUNDUP(sz);

  uint64 sz1;

  if ((sz1 = uvmalloc(pagetable, sz, sz + 2 \* PGSIZE)) == 0)

    goto bad;

  sz = sz1;

  uvmclear(pagetable, sz - 2 \* PGSIZE);

  sp = sz;

  stackbase = sp - PGSIZE;

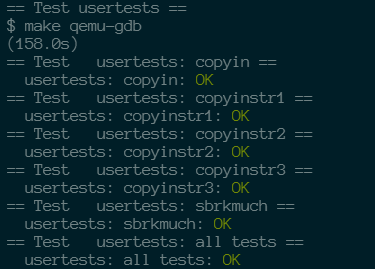
  if (uvm2k(pagetable, p->kpagetable, 0, sz) < 0)

    goto bad;

  ...

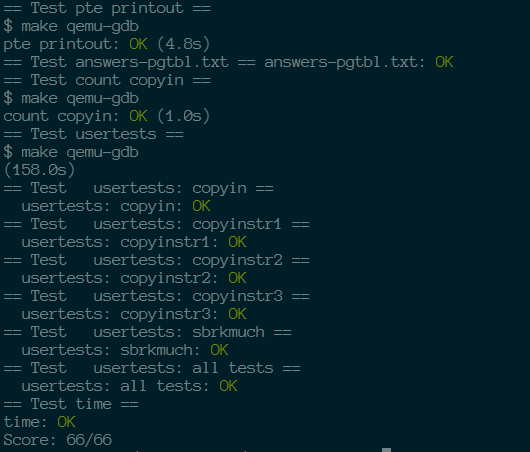
}

（4）运行simplify copyin/copyinstr



**3.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**3.5 实验小结**

本次实验的完成较为困难，它通过修改内核完成proc维护每个进程的内核页表并在切换进程时切换内核页表，并且实现了用户页表到内核页表的映射。页表作为操作系统中内存管理的重要部分，实现了将虚拟地址转换为物理地址的重要功能，这次实验提高了我对页表结构及其应用的认识和熟悉程度。