# Lab5: pgtbl

- 姓名：吴欣怡
- 学号：PB21051111
- 虚拟机用户名：OS-PB21051111

# Print a page table

## 实验分析

实现vmprint() 的函数，接收一个以下面描述的格式打印该页表。参考freewalk函数，可以知道pte、PTE_V、PTE_R、PTE_W、PTE_X的大致含义。参考输出的格式，分析知采用递归输出。

## 实验过程

参考实验文档中的提示：
在defs.h中添加函数声明

```
int             vmprint(pagetable_t pagetable);
```

在 exec.c 返回前打印页表。

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

阅读vm.c中的freewalk函数：
pte 表示页表项，而 PTE2PA(pte) 是将页表项转换为物理地址的函数。
（pte & PTE_V）时页表有效，
(pte & (PTE_R|PTE_W|PTE_X)) == 0时此页表没有读、写和执行权限，也就是说这个页表不是一个实际存储数据的页，应当指向一个较低级别的页表。
容易知道需要递归来打印，在freewalk的基础上作一定的修改，得到print函数（由于实验文档中说明了vmprint只接收pagetable_t参数，所以需要另外写一个会传递depth的函数来调用）：
首先页表需要有效才会开始打印操作，先按照dpeth大小来输出".."
然后输出当前页表的PTE索引、PTE比特位以及从PTE提取的物理地址。
再判断当前页表是否有子节点，若有则调用print((pagetable_t) child,depth+1)，继续输出下一层。这个递归在页表不再有效时会终止。
vmprint中打印page table信息并直接调用print即可。

```
int print(pagetable_t pagetable,int depth){
  for(int i = 0; i < 512; i++){
      pte_t pte = pagetable[i];
      if(pte & PTE_V) {//有效
        printf("..");
        for(int j=0;j<depth;j++)
          printf(" ..");
```

```
        printf("%d: pte %p pa %p\n",i,pte,PTE2PA(pte));
        if((pte & (PTE_R|PTE_W|PTE_X)) == 0){//有子节点
          uint64 child = PTE2PA(pte);
          //depth=depth+1;
          print((pagetable_t) child,depth+1);
        }
      }
  }
  return 0;

}

int vmprint(pagetable_t pagetable)
{
  printf("page table %p\n",pagetable);
  return print(pagetable,0);
}
```

# A kernel page table per process

## 实验分析

让每一个进程进入内核态后，都能有自己的独立内核页表。
修改proc结构体，修改kvminit并在allocproc中调用。
把procinit的功能迁移到allocproc。修改修改
scheduler()和freeproc。

## 实验过程

### 第一步（vm.c部分）

(1)在kernel/proc.h的proc结构体中添加一个存储进程专享的内核态页表

```
pagetable_t kernelpgtbl;     // Kernel page table
```

首先看到原始的kvminit中多次调用kvmmap，原kvmmap的用法是要将虚拟地址 va 到 va + sz 的范围映射到物理地址 pa 到 pa + sz 的范围，权限为 perm。
修改kvmmap，增加参数pgtbl，因为不再统一用于一个页表。

```
void
kvmmap(pagetable_t pgtbl, uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(pgtbl, va, sz, pa, perm) != 0)
    panic("kvmmap");
}
```

修改kvminit，把原来kvminit的功能拆成kvminit()和kvminit_newpgtbl()，以实现对进程的页表和全局内核页表的不同定义。
对应地修改kvm_map_pagetable（把kvmmap按照新的定义使用）和kvmpa。

```
void kvm_map_pagetable(pagetable_t pgtbl) {
  // 将各种内核需要的 direct mapping 添加到页表 pgtbl 中。

  kvmmap(pgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  kvmmap(pgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  kvmmap(pgtbl, CLINT, CLINT, 0x10000, PTE_R | PTE_W);

  kvmmap(pgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  kvmmap(pgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

  kvmmap(pgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);

  kvmmap(pgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
}

pagetable_t
kvminit_newpgtbl()
{
  pagetable_t pgtbl = (pagetable_t) kalloc();
  memset(pgtbl, 0, PGSIZE);

  kvm_map_pagetable(pgtbl);

  return pgtbl;
}

/*
 * create a direct-map page table for the kernel.
 */
void
kvminit()
{
  kernel_pagetable = kvminit_newpgtbl();
}

// kvmpa 将内核逻辑地址转换为物理地址（添加第一个参数 kernelpgtbl）
uint64
kvmpa(pagetable_t pgtbl, uint64 va)
{
  uint64 off = va % PGSIZE;
  pte_t *pte;
  uint64 pa;

  pte = walk(pgtbl, va, 0);
```

```
  if(pte == 0)
    panic("kvmpa");
  if((*pte & PTE_V) == 0)
    panic("kvmpa");
  pa = PTE2PA(*pte);
  return pa+off;
}
```

## 第二步

修改procinit和allocproc
在创建进程的时候，为进程分配独立的内核页表(kvminit_newpgtbl();)以及内核栈, 将内核栈映射到固定的逻辑地址上;

```
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;
  }

  // An empty user page table.
  p->pagetable = proc_pagetable(p);
  if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  p->kernelpgtbl = kvminit_newpgtbl();

  char *pa = kalloc();
  if(pa == 0)
```

```
    panic("kalloc");
  uint64 va = KSTACK((int)0);
  kvmmap(p->kernelpgtbl, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
  p->kstack = va;


  // 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;
}
```

修改scheduler()，加载进程的内核页表到核心的satp 寄存器，只有进程在执行的过程中需要进入进程自己的内存页表，一个进程运行结束后就调整到kernel_pagetable，这样就能满足没有进程运行时scheduler() 应当使用kernel_pagetable的条件。所以只需要修改进程进入从可行进入运行的过程中的操作就行：

```
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;

        //切换到进程自己的内核页表，并调用sfence_vma
        w_satp(MAKE_SATP(p->kernelpgtbl));
        sfence_vma();

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

        kvminithart();

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
```

```
        found = 1;
      }
      release(&p->lock);
    }
#if !defined (LAB_FS)
    if(found == 0) {
      intr_on();
      asm volatile("wfi");
    }
#else
    ;
#endif
  }
}
```

修改freeproc，在vm.c中新建一个函数来实现这个功能：
该页表项指向更低一级的页表，则递归释放低一级页表及其页表项
与之相关的页表项都释放之后，就释放当前页占用的内存空间

```
void
kvm_free_kernelpgtbl(pagetable_t pagetable)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    uint64 child = PTE2PA(pte);
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      kvm_free_kernelpgtbl((pagetable_t)child);
      pagetable[i] = 0;
    }
  }
  kfree((void*)pagetable);
}
```

# Simplify　copyin/copyinstr（hard）

## 实验分析

为每个进程的内核页表添加用户地址映射
在内核更改进程的用户映射的每一处，都以相同的方式更改进程的内核页
表。包括fork() , exec() , 和sbrk() 。注意（0 到 PLIC 段）地址空间的映射同步。

## 实验步骤

在defs.h中声明
PGROUNDUP 是一个用于将地址向上对齐到页面边界的宏或函数。在上述代码中，PGROUNDUP 的作用是确保虚拟地址（start）是页面大小（PGSIZE）的整数倍。

```
int
```

```
kvmcopymappings(pagetable_t src, pagetable_t dst, uint64 start, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;

  // PGROUNDUP: 对齐页边界，防止 remap
  for(i = PGROUNDUP(start); i < start + sz; i += PGSIZE){
    if((pte = walk(src, i, 0)) == 0)
      panic("kvmcopymappings: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("kvmcopymappings: page not present");
    pa = PTE2PA(*pte);
    // `& ~PTE_U` 表示将该页的权限设置为非用户页
    flags = PTE_FLAGS(*pte) & ~PTE_U;
    if(mappages(dst, i, PGSIZE, pa, flags) != 0){
      goto err;
    }
  }

  return 0;

 err:
  uvmunmap(dst, 0, i / PGSIZE, 0);
  return -1;
}

// 将程序内存从 oldsz 缩减到 newsz
// 用于内核页表内程序内存映射与用户页表程序内存映射之间的同步
uint64
kvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
{
  if(newsz >= oldsz)
    return oldsz;

  if(PGROUNDUP(newsz) < PGROUNDUP(oldsz)){
    int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;
    uvmunmap(pagetable, PGROUNDUP(newsz), npages, 0);
  }

  return newsz;
}
```

再次修改kvminit()，单独给全局内核页表映射 CLINT，避免进程内核页表 CLINT 与程序内存映射冲突。

```
void kvm_map_pagetable(pagetable_t pgtbl) {

  // uart registers
  kvmmap(pgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
```

```
  kvmmap(pgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // CLINT
  // kvmmap(pgtbl, CLINT, CLINT, 0x10000, PTE_R | PTE_W);

  // PLIC
  kvmmap(pgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  // ......
}


void
kvminit()
{
  kernel_pagetable = kvminit_newpgtbl();
  // CLINT *is* however required during kernel boot up and
  // we should map it for the global kernel pagetable
  kvmmap(kernel_pagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
}
```

用于管理内核中的虚拟内存，确保内核中的页表状态与用户空间中的页表状态保持同步，同时避免实际释放内存。
修改exec.c中的exec()函数，增加对内存空间PLIC的判断，修改内核更改进程的用户映射的部分。

```
int
exec(char *path, char **argv)
{
......
// Load program into memory.
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    uint64 sz1;
    if(sz1>=PLIC){
      goto bad;
    }
    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    sz = sz1;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
```

```
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  .......
  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(p->name, last, sizeof(p->name));

  // 清除内核页表中对程序内存的旧映射，然后重新建立映射。
  uvmunmap(p->kernelpgtbl,0,PGROUNDUP(oldsz)/PGSIZE,0);
  kvmcopymappings(pagetable,p->kernelpgtbl,0,sz);

  // Commit to the user image.
  oldpagetable = p->pagetable;
  p->pagetable = pagetable;
  p->sz = sz;
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer
  proc_freepagetable(oldpagetable, oldsz);
  .......
}
```

修改userinit(),同步程序内存映射到进程内核页表中:

```
// kernel/proc.c
void
userinit(void)
{
  // ......

  // allocate one user page and copy init's instructions
  // and data into it.
  uvminit(p->pagetable, initcode, sizeof(initcode));
  p->sz = PGSIZE;
  kvmcopymappings(p->pagetable, p->kernelpgtbl, 0, p->sz); // 同步程序内存映射到进程内核
页表中

  // ......
}
```

修改fork():

```
  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0 || kvmcopymappings(np-
>pagetable,np->kernelpgtbl,0,p->sz)){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  np->parent = p;
```

修改growproc()，因为在sbrk()中调用了这个函数:

```
 int
growproc(int n)
{
  uint sz;
  struct proc *p = myproc();

  sz = p->sz;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0 ) {
      return -1;
    }
    if(kvmcopymappings(p->pagetable, p->kernelpgtbl, sz, n) != 0) {
      uvmdealloc(p->pagetable, newsz, sz);
      return -1;
    }
    sz = newsz;

  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }
  p->sz = sz;
  return 0;
}
```

修改copyin、copyout的定义（及声明）

```
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);
}
```

## 实验评分

```
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ python3 grade-lab-p
gtbl
make: “kernel/kernel”已是最新。
== Test pte printout == pte printout: OK (2.0s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin == count copyin: OK (1.2s)
== Test usertests == (267.3s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 66/66
```

## 实验总结

中间没有给CLINET单独的映射权限，没有注意到用户地址的PTE在进程的内核页表中的权限问题，走了很多弯路。