Lab: page tables

Print a page table (easy)

任务描述

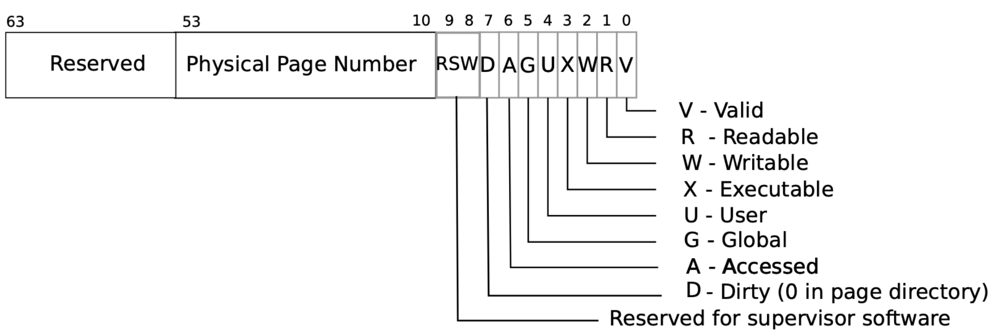
在第一个进程结束的时候，打印页表。

需要实现一个vmprint()函数来实现打印页表。

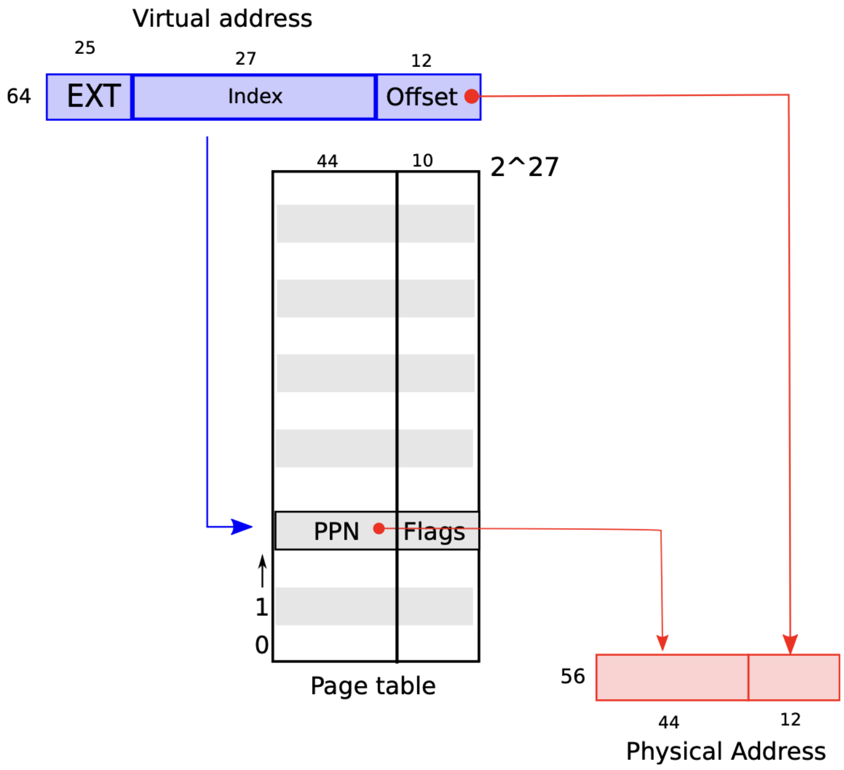
```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

原理解析

PTE:page table entries(页表指向)



VA:virtual address(虚拟地址)



PA:Physical address(真实地址)

指向硬件内存的64位整数

Pagetable(页表)

页表本身是一个PA。

它是一个指针，指向一个大小固定为4kb的内存块的开头。

页表内可以存储真正的数据，也可以存储页表。

页表是64位整数，所以一页内存可以存下512个页表。

从VA映射到PA的流程

每个进程有一个专属的proc结构

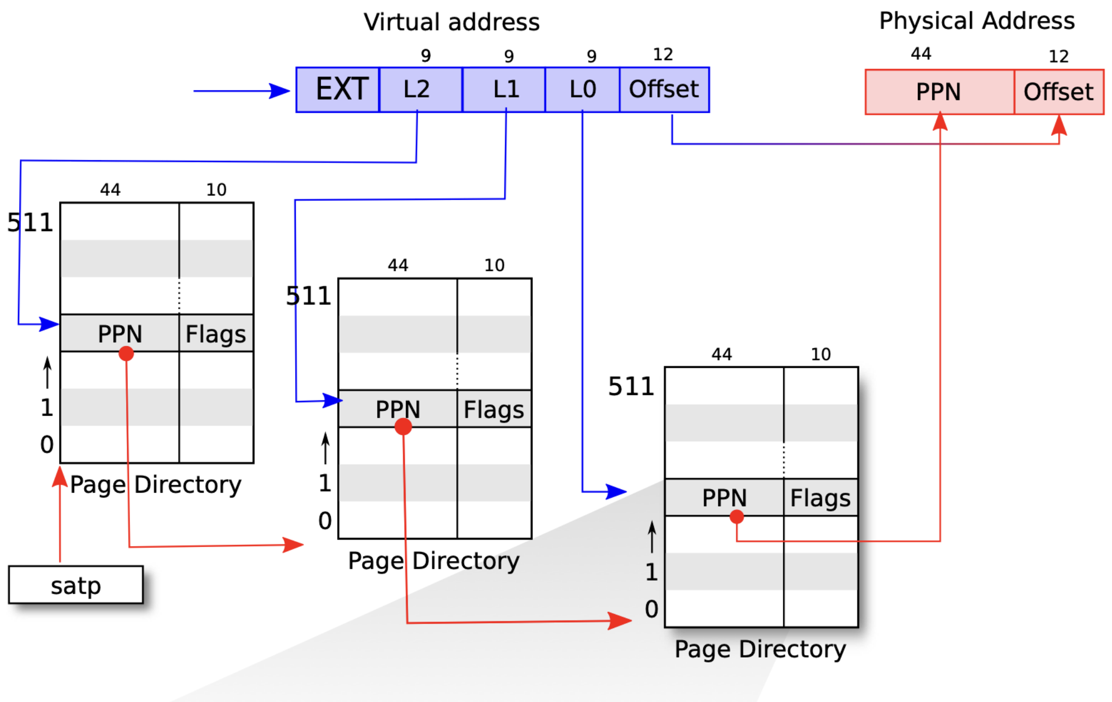
里面有进程专属的pagetable

VA映射到PA要经过三次转化。

VA中的index有27位，分三段来指向页表。

页表指向真是内存的结构分成了三层。

VA->proc pagetable->L2 pagetable->L1 pagetblae->L0 pageable->PA



样例解析

```
… 0:
… … 0:
… … … 0: 标志位0x1f, 用户可以访问
… … … 1: 标志位0x0f, 用户不可以访问
… … … 2: 标志位0x1f, 用户可以访问
… 255:
… … 511:
… … … 510: 标志位0x07,可读写
… … … 511: 标志位0x0b, 可执行可读
```

实操

在defs.h中加入

```
void            vmprint(pagetable_t);
```

在vm.c中加入

```
void vmprintlevel(pagetable_t x,int level){
  for(int i=0;i<512;i++){
    pte_t pte=x[i];
    pagetable_t child=(pagetable_t)PTE2PA(pte);
    if((pte&PTE_V)==0)continue;
    for(int i=0;i<level;i++)printf(".. ");
    printf("..%d: pte %p pa %p\n",i,pte,child);
    if((pte&(PTE_W|PTE_R|PTE_X))==0)vmprintlevel(child,level+1);
  }
}
void
vmprint(pagetable_t x){
  printf("page table %p\n",x);
  vmprintlevel(x,0);
```

```
}
```

A kernel page table per process (hard)

任务描述

往每个进程中复制一份内核页表

任务步骤

1.在**kernel/proc.h**为**truct proc**加入内核页表。

```
pagetable_t krnl_pagetable;
```

2.**kernel/vm.c**中**vmkinit**是为最开始的全局**kernel_pagetable**创建初始化的，将其修改为可对某个页表进行初始化的函数。**CLINT**只会在开始是被加载，所以单独在**procinit**函数里进行初始化。同时修改**kvmmap**和**kvmpa**的传参。并且修改对应的函数使用的传参。

```
void
kvmmap(pagetable_t now_pagetable,uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(now_pagetable, va, sz, pa, perm) != 0)
    panic("kvmmap");
}
void
auto_kvmmap(pagetable_t now_pagetable){
  // uart registers
  kvmmap(now_pagetable,UART0, UART0, PGSIZE, PTE_R | PTE_W);
  // virtio mmio disk interface
  kvmmap(now_pagetable,VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
  // CLINT
//  kvmmap(now_pagetable,CLINT, CLINT, 0x10000, PTE_R | PTE_W);
  // PLIC
  kvmmap(now_pagetable,PLIC, PLIC, 0x400000, PTE_R | PTE_W);
  // map kernel text executable and read-only.
  kvmmap(now_pagetable,KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R |
PTE_X);
  // map kernel data and the physical RAM we'll make use of.
  kvmmap(now_pagetable,(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(now_pagetable,TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
}
void
kvminit()
{
  kernel_pagetable = (pagetable_t) kalloc();
  memset(kernel_pagetable, 0, PGSIZE);
  kvmmap(kernel_pagetable,CLINT, CLINT, 0x10000, PTE_R | PTE_W);
  auto_kvmmap(kernel_pagetable);
}
uint64
kvmpa(uint64 va,pagetable_t now_pagetable)
{
  uint64 off = va % PGSIZE;
```

```
  pte_t *pte;
  uint64 pa;

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

3.修改**kernel/proc.h**中**procinit**里内核栈空间的分配，将其改到**allocproc**里。

```
void
procinit(void)
{
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");

      // Allocate a page for the process's kernel stack.
      // Map it high in memory, followed by an invalid
      // guard page.
/*      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;*/
  }
//  kvminithart();
}
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->krnl_pagetable = (pagetable_t) kalloc();
  memset(p->krnl_pagetable,0,PGSIZE);
  auto_kvmmap(p->krnl_pagetable);
  char *pa = kalloc();
  if(pa == 0)panic("kalloc");
  uint64 va = KSTACK((int) (p - proc));
```

```
  kvmmap(p->krnl_pagetable,va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
  p->kstack = va;
  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;
  }

  // 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;
}
```

4.在**schduler**添加页表切换，切换进程的时候切换页表。

```
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;
        w_satp(MAKE_SATP(p->krnl_pagetable));
        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
  }
}
```

5.在**freeproc**里释放掉内核页表。

先断链接再清空。

```
void
auto_ukvmmap(pagetable_t now_pagetable){
  // uart registers
  //kvmmap(now_pagetable,UART0, UART0, PGSIZE, PTE_R | PTE_W);
  uvmunmap(now_pagetable, UART0, 1, 0);
  // virtio mmio disk interface
  //kvmmap(now_pagetable,VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
  uvmunmap(now_pagetable,VIRTIO0, 1, 0);
  // CLINT
  //kvmmap(now_pagetable,CLINT, CLINT, 0x10000, PTE_R | PTE_W);
  //uvmunmap(now_pagetable,CLINT, 0x10000/PGSIZE, 0);
  // PLIC
  //kvmmap(now_pagetable,PLIC, PLIC, 0x400000, PTE_R | PTE_W);
  uvmunmap(now_pagetable,PLIC, 0x400000/PGSIZE, 0);
  // map kernel text executable and read-only.
  //kvmmap(now_pagetable,KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R |
PTE_X);
  uvmunmap(now_pagetable,KERNBASE, PGROUNDUP((uint64)etext-KERNBASE)/PGSIZE, 0);
  // map kernel data and the physical RAM we'll make use of.

  //kvmmap(now_pagetable,(uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext,
PTE_R | PTE_W);
  uvmunmap(now_pagetable,(uint64)etext, PGROUNDUP((PHYSTOP-
(uint64)etext))/PGSIZE, 0);
  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  //kvmmap(now_pagetable,TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
  uvmunmap(now_pagetable,TRAMPOLINE, 1 , 0);
}
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->krnl_pagetable){
    auto_ukvmmap(p->krnl_pagetable);
```

```
    uvmunmap(p->krnl_pagetable,0,PGROUNDUP(p->sz)/PGSIZE,0);
  //kfree((void*)kvmpa(p->kstack,p->krnl_pagetable));
    uvmunmap(p->krnl_pagetable,p->kstack,1,1);
  //printf("yes\n");
    uvmfree(p->krnl_pagetable,0);
  }
//  vmprint(p->krnl_pagetable);
//  uvmfree(p->krnl_pagetable, 0);
  p->krnl_pagetable=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;
}
```

Simplify `copyin/copyinstr` (hard)

任务描述

让`copyin/copyinstr`函数去内存的时候不需要再用pagetable而是直接寻址，加快`copyin/copyinstr`的效率。

实现方法：将进程user态的内存复制一份到进程的k_pagetable里，就可以在kernel中直接解引用。

任务步骤

1.修改`copyin/copyinstr`为更新后的函数

```
int
copyin_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len);
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  return copyin_new(pagetable,dst,srcva,len);
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(srcva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (srcva - va0);
    if(n > len)
      n = len;
    memmove(dst, (void *)(pa0 + (srcva - va0)), n);

    len -= n;
    dst += n;
    srcva = va0 + PGSIZE;
```

```
  }
  return 0;
}

// Copy a null-terminated string from user to kernel.
// Copy bytes to dst from virtual address srcva in a given page table,
// until a '\0', or max.
// Return 0 on success, -1 on error.
int
copyinstr_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max);
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  return copyinstr_new(pagetable,dst,srcva,max);

  uint64 n, va0, pa0;
  int got_null = 0;

  while(got_null == 0 && max > 0){
    va0 = PGROUNDDOWN(srcva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (srcva - va0);
    if(n > max)
      n = max;

    char *p = (char *) (pa0 + (srcva - va0));
    while(n > 0){
      if(*p == '\0'){
        *dst = '\0';
        got_null = 1;
        break;
      } else {
        *dst = *p;
      }
      --n;
      --max;
      p++;
      dst++;
    }

    srcva = va0 + PGSIZE;
  }
  if(got_null){
    return 0;
  } else {
    return -1;
  }
}
```

2.增加函数**vmcopypage**。

```
void
vmcopypage(pagetable_t pagetable,pagetable_t krnl_pagetable,uint64 start,uint64
sz){
  for(uint64 i=start;i<start+sz;i+=PGSIZE){
    pte_t* pte=walk(pagetable,i,0);
    pte_t* krnl_pte=walk(krnl_pagetable,i,1);
    if(!pte||!krnl_pte){
      panic("vmcopypage");
    }
    *krnl_pte=(*pte)&~(PTE_U|PTE_U|PTE_X);
  }
}
```

3.在**sys_sbrk**，**userinit**，**fork**，**exec**中都增加对user页表的复制。

exec中要释放掉旧用户的user页表，并且加上**PLIC**的限制。

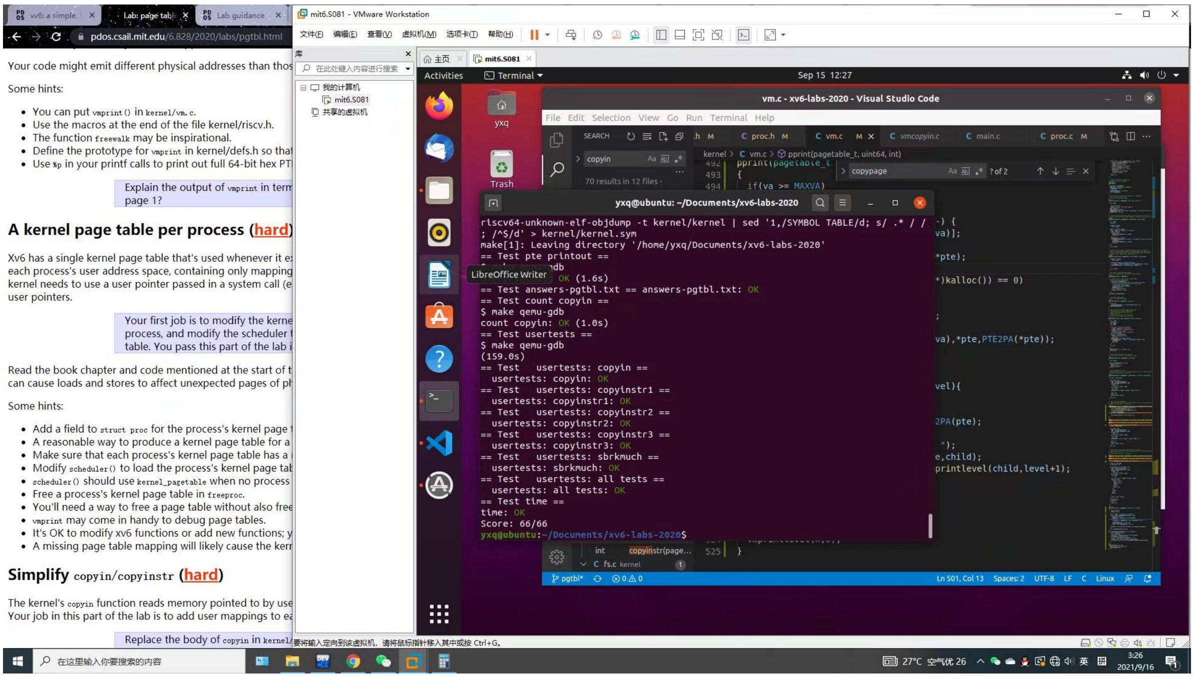
```
uint64
sys_sbrk(void)
{
  int addr;
  int n;
  struct proc *p=myproc();
  if(argint(0, &n) < 0)
    return -1;
  addr = p->sz;
  if(growproc(n) < 0)
    return -1;
  if(n>0){
    vmcopypage(p->pagetable,p->krnl_pagetable,addr,n);
  }else{
    for(int j=addr-PGSIZE;j>=addr+n;j-=PGSIZE){
      uvmunmap(p->krnl_pagetable,j,1,0);
    }
  }
  return addr;
}
void
userinit(void)
{
  struct proc *p;

  p = allocproc();
  initproc = p;

  // allocate one user page and copy init's instructions
  // and data into it.
  uvminit(p->pagetable, initcode, sizeof(initcode));
  p->sz = PGSIZE;
  vmcopypage(p->pagetable,p->krnl_pagetable,0,p->sz);
  // prepare for the very first "return" from kernel to user.
  p->trapframe->epc = 0;      // user program counter
  p->trapframe->sp = PGSIZE;  // user stack pointer

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");
```

```
  p->state = RUNNABLE;

  release(&p->lock);
}
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;
  vmcopypage(np->pagetable,np->krnl_pagetable,0,np->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);

  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)
{
  char *s, *last;
  int i, off;
  uint64 argc, sz = 0, sp, ustack[MAXARG+1], stackbase;
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pagetable_t pagetable = 0, oldpagetable;
```

```c
  struct proc *p = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);

  // Check ELF header
  if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pagetable = proc_pagetable(p)) == 0)
    goto bad;

  // 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 = 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;
    if(sz1>=PLIC)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  p = myproc();
  uint64 oldsz = p->sz;

  // Allocate two pages at the next page boundary.
  // Use the second as the user stack.
  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;
```

```c
  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
      goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[argc] = sp;
  }
  ustack[argc] = 0;

  // push the array of argv[] pointers.
  sp -= (argc+1) * sizeof(uint64);
  sp -= sp % 16;
  if(sp < stackbase)
    goto bad;
  if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
    goto bad;
  uvmunmap(p->krnl_pagetable,0,PGROUNDUP(oldsz)/PGSIZE,0);
  vmcopypage(pagetable,p->krnl_pagetable,0,sz);
  // arguments to user main(argc, argv)
  // argc is returned via the system call return
  // value, which goes in a0.
  p->trapframe->a1 = sp;

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

  // 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);
  if(p->pid==1)vmprint(p->pagetable);
  return argc; // this ends up in a0, the first argument to main(argc, argv)

 bad:
  if(pagetable)
    proc_freepagetable(pagetable, sz);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

Grade


mit6.S081 - VMware Workstation
pdos.csail.mit.edu/6.828/2020/labs/pgtbl.html
Your code might emit different physical addresses than thos
Some hints:
You can put vmprint() in kernel/vm.c.
Use the macros at the end of the file kernel/riscv.h.
The function freewalk may be inspirational.
A kernel page table per process (hard)
Simplify copyin/copyinstr (hard)
vm.c - xv6-labs-2020 - Visual Studio Code
yxq@ubuntu: ~/Documents/xv6-labs-2020
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/yxq/Documents/xv6-labs-2020'
== Test pte printout ==
OK (1.6s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.0s)
== Test usertests ==
$ make qemu-gdb
(159.0s)
== 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
yxq@ubuntu:~/Documents/xv6-labs-2020$
27°C 空气优 26
3:26
2021/9/16