

实验四 Page tables

PB20050987 梁兆懿

在实验4中，我阅读了xv6的第三章，学习了有关页表的知识。在实验四中，我们将探索页表，并且实现页表的分配以及简化功能。

实验需要切换分支。

```
ubuntu@VM5878-LZY:/home/ubuntu/文档/xv6-labs-2020$ git fetch
ubuntu@VM5878-LZY:/home/ubuntu/文档/xv6-labs-2020$ git checkout pgtbl
```

Print a page table

第一个实验需要实现打印页表的功能。该函数需要被命名为 `vmprint()`。在该函数实现后，页表打印出来的结果应该如下：

```
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
```

实验代码

根据提示，我们看到 `kernel/riscv.h` 的宏定义以及 `kernel/vm.c` 的 `freewalk` 函数，可以知道页表的数据结构为 `pagetable_t`，是一个 `uint64` 指针。页表是一个有多级PTE的数组。

因此，我们可以根据如上信息，参照 `vm.c` 中的 `freewalk` 函数，实现打印功能：

```
void
_pteprint(pagetable_t pagetable, int level)
{
    for(int i = 0; i < 512; i++) {
        pte_t pte = pagetable[i];

        if (pte & PTE_V) {
            for (int j = 0; j <= level; j++)
                printf(".. ");
            printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
        }
        if ((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_W)) == 0) {
            uint64 child = PTE2PA(pte);
            _pteprint((pagetable_t)child, level+1);
        }
    }
}
```

```
void
vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    _pteprint(pagetable, 0);
}
```

为了在 exec.c 中引用 vmprint 函数，我们需要在 defs.h 文件添加原型：

```
//vm.c
void _pteprint(pagetable_t pagetable, int level);
void vmprint(pagetable_t pagetable);
```

根据提示，在 exec.c 中，我们添加如下代码实现打印第一个进程的页表信息：

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

此外，对于思考题 Explain the output of `vmprint` in terms of Fig 3-4 from the text. What does page 0 contain? What is in page 2? When running in user mode, could the process read/write the memory mapped by page 1?

我们将答案保存到 `answer-pgtbl.txt` 中，具体为

```
The size of the content in the LOAD can be put into a page, and page0 should map
the content in this LOAD, which should be the code segment and the data segment.
Page2 is the process stack. Page1 is the guard page, looking at uvmclear to find
that the guard page's PTE_U is set to 0. User-mode code access hardware will
produce a page fault exception, so it couldn't be read.
```

实验结果

```
ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
nel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/
proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/
sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o
kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o k
ernel/virtio_disk.o kernel/vmcopyin.o kernel/stats.o kernel/sprintf.o
riscv64-linux-gnu-ld: 警告: 无法找到项目符号 _entry; 缺省为 0000000080000000
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > kernel/kernel.sym
make[1]: 离开目录"/home/ubuntu/文档/xv6-labs-2020"
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (4.2s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
```

A kernel page table per process

该实验需要为每一个进程在内核执行时都有自己的内核页表。具体为维护每个进程的内核页表，修改调度程序以在切换进程时切换内核页表，每个进程内核页表应与现有全局内核页表相同。

实验代码

首先，根据提示，我们在 proc.h 中为进程内核页表添加一个字段：

```
struct proc{
    pagetable_t proc_k_pt; //kernel pagetable
}
```

随后，我们参考kvminit函数，创建函数ukvminit，ukvmmap，对struct proc中的proc_k_pt赋值，从而创建内核页表：

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

pagetable_t ukvminit()
{
    pagetable_t kpagetable = (pagetable_t)kalloc();
    memset(kpagetable, 0, PGSIZE);
    ukvmmap(kpagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
    ukvmmap(kpagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
    ukvmmap(kpagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
    ukvmmap(kpagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
    ukvmmap(kpagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R |
PTE_X);
    ukvmmap(kpagetable, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R
| PTE_W);
    ukvmmap(kpagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
    return kpagetable;
}
```

在defs.h中，添加函数的声明：

```
void ukvmmap(pagetable_t kpagetable, uint64 va, uint64 pa, uint64 sz, int perm);
pagetable_t ukvminit();
```

我们在进程中调用该函数，为proc_k_pt赋值。参照为用户页表赋值的代码，我们在allocproc函数中容易写出创建内核页表的代码：

```
static struct proc* allocproc(void){

    p->pagetable = proc_pagetable(p);
    if(p->pagetable == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    } //参照为用户页表赋值的代码

    p->proc_k_pt = ukvminit();
    if(p->proc_k_pt == 0){
        freeproc(p);
    }
```

```

        release(&p->lock);
        return 0;
    }

}

```

完成内核页表的创建后，根据提示，我们要为每个用户进程页表分配一个内核栈，需要注释掉procinit函数中内核栈的创建，并且在allocproc函数中为每一个进程创建一个独立的进程内核页表：

```

// initialize the proc table at boot time.
void
procinit(void)
{
    struct proc *p;jiang

    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));
        //kvmmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
        //p->kstack = va;
        //删除原有内核栈创建代码
    }
    kvmminithart();
}

```

为每一个进程创建一个独立的进程内核页表：

```

static struct proc* allocproc(void){

    char *pa = kalloc();
    if(pa == 0)
        panic("kalloc");
    uint64 va = KSTACK((int) (p - proc));
    ukvmmmap(p->proc_k_pt,va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
    p->kstack = va;

}

```

为了确保在切换进程时能够将对应进程的用户内核页表的地址载入SATP寄存器中，我们需要修改kernel/proc.c的scheduler函数，完成进程的调度：

```

void scheduler(void)
{

    w_satp(MAKE_SATP(p->proc_kernel_pagetable));
    sfence_vma();
    swtch(&c->context, &p->context);
    kvminithart();

}

```

根据实验提示，我们需要在销毁进程时释放内存页表。参考 freewalk，但是直接调用freewalk会报错，原因是把真实的物理地址也释放了。因此我们需要根据freewalk 函数重新写一个释放释放进程内核页表的函数：

```

void proc_freekpt(pagetable_t pagetable){
    for (int i = 0; i < 512; ++i) {
        pte_t pte = pagetable[i];
        if ((pte & PTE_V)) {
            pagetable[i] = 0;
            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
                uint64 child = PTE2PA(pte);
                proc_freekpt((pagetable_t)child);
            }
        } else if (pte & PTE_V) {
            panic("proc free kernelpagetable : leaf");
        }
    }
    kfree((void*)pagetable);
}

```

修改freeproc，添加释放内核页表的代码：

```

static void freeproc(struct proc *p)
{
    if(p->trapframe)
        kfree((void*)p->trapframe);
    p->trapframe = 0;

    if (p->kstack) {
        pte_t* pte = walk(p->p_k_pt, 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->p_k_pt)
        proc_freekpt(p->p_k_pt);

    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;
}

```

最后，在 vm.c 有个 kvmpa，改成用当前进程的内核页表的，导入 proc.h，从 myproc 获取内核页表：

```

#include "spinlock.h"
#include "proc.h"

//pte = walk(kernel_pagetable, va, 0);
struct proc *p = myproc();
pte = walk(p->proc_k_pt, va, 0);

```

实验结果

```

ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
pte printout: OK (4.7s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.8s)
== Test usertests ==
$ make qemu-gdb
(291.9s)
== 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
ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020$

```

Simplify copyin/copyinstr

内核的copyin函数读取用户指针指向的内存。实验需要将用户映射添加到每个进程的内核页表(在上一节中创建)，从而允许copyin直接取消引用用户指针。

实验代码

根据实验提示，我们引用 copyin_new 和 copyinstr_new 代替原有的 copyin 和 copyinstr。并且添加相关头声明实现该功能。

在 def.s 中：

```

int copyin_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len);
int copyinstr_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max);

```

对copyin 和 copyinstr的修改:

```
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);
}
```

由于函数 `fork()`, `exec()` 和 `sbrk()` 会修改进程内核页表的大小, 我们需要对其进行修改。

首先我们写一个函数把用户的进程页表映射到内核页表:

```
uvmcopy_n_p(pagetable_t old, pagetable_t new, uint64 begin, uint64 end)
{
    pte_t *pte, *newPte;
    uint64 pa, i;
    uint flags;

    for(i = PGROUNDDOWN(begin); i < end; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
            panic("uvmcopy_n_p: pte should exist");
        if((newPte = walk(new, i, 1)) == 0)
            panic("uvmcopy_n_p: page not present");
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte) & (~PTE_U);

        *newPte = PA2PTE(pa) | flags;
    }
    return 0;
}
```

`fork` 里需要添加调用

```
if (uvmcopy_n_p(p->pagetable, p->kpagetable, p->sz) != 0)
    return -1;
```

`exec` 里也需要调用, 注意调用前需要添加检测, 防止程序大小超过 PLIC, 释放旧的内核页表里映射的用户页表。

```
if(sz1 >= PLIC)//防止程序大小超过 PLIC
    goto bad;

oldpagetable = p->pagetable;
p->pagetable = pagetable;

uvmunmap(p->proc_k_pt, 0, PGROUNDDOWN(p->sz)/PGSIZE, 0);//释放
uvmcopy_n_p(pagetable, p->proc_k_pt, 0, sz);
```

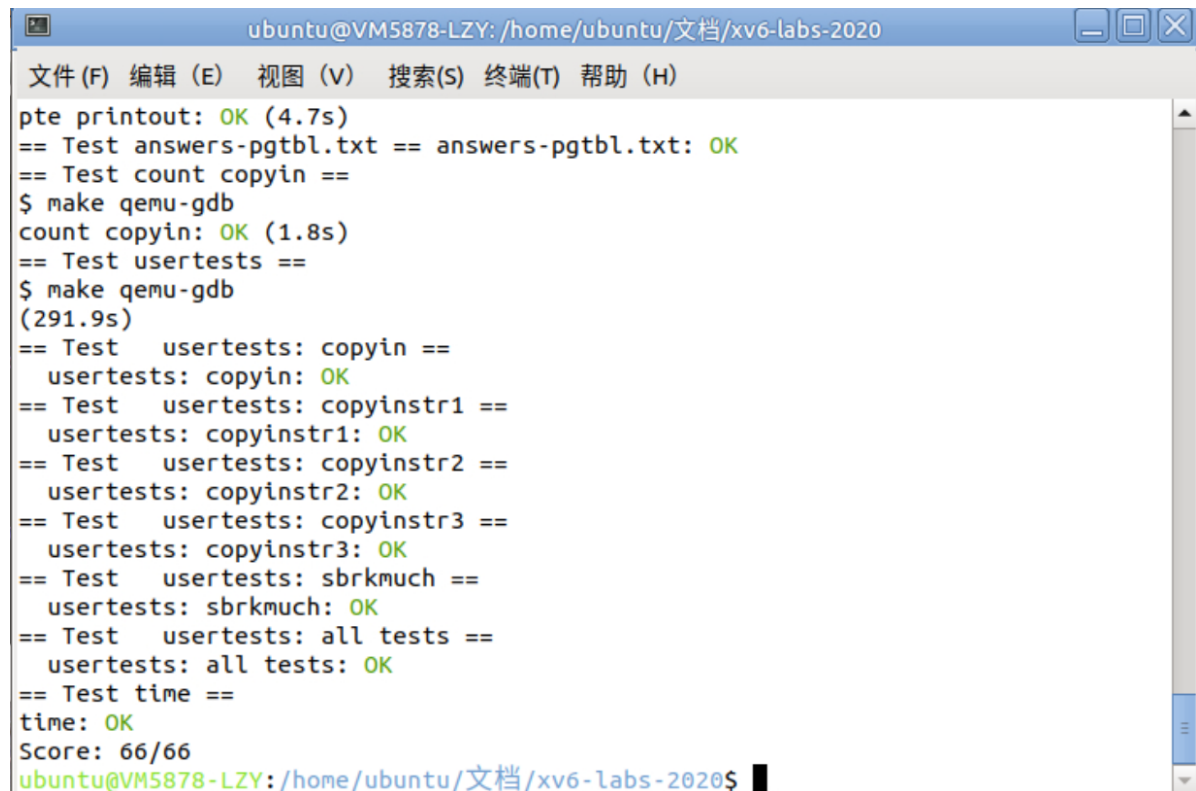
根据提示, `sys_sbrk()`函数会修改进程内核页表, 通过函数`growproc`实现。因此我们修改该函数, 缩小 `kernel_pagetable` 的相应映射。

```
int new = p->sz + n;
if(PGROUNDDOWN(newsz) < PGROUNDUP(p->sz))
{
    int npages = (PGROUNDUP(p->sz) - PGROUNDUP(new)) / PGSIZE;
    uvmunmap(p->proc_k_pt, PGROUNDUP(new), npages, 0);
}
```

在 `userinit` 的内核页表中包含第一个进程的用户页表:

```
uvmcopy_n_p(p->pagetable, p->proc_k_pt, 0, p->sz);
```

实验结果



```
ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
pte printout: OK (4.7s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.8s)
== Test usertests ==
$ make qemu-gdb
(291.9s)
== 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
ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020$
```

思考题

Explain why the third test `srcva + len < srcva` is necessary in `copyin_new()`: give values for `srcva` and `len` for which the first two test fail (i.e., they will not cause to return -1) but for which the third one is true (resulting in returning -1).

答: 原因是需要防止溢出, 例如 `(ph.vaddr + ph.memsz < ph.vaddr)` 检查总和是否溢出 64 位整数。

实验收获与感受

这次实验难度较大, 写了整整两天时间。在写实验时不知道实验二和实验三的测试顺序调换了, 也不知道实验二测试时间很长。因此在 `make grade` 过程中, 第二个测试没过, 第三个测试没出结果, 我以为是实验代码出了问题, 修改了一整天...最后偶然一次测试时, 把测试程序挂在虚拟机上五分钟, 发现居然通过了T_T。以后做实验还是需要细心。此外, 在本次实验中, 我对页表相关知识以及其结构有了更加深刻的理解, 做完实验满满都是成就感, 收获颇丰。

