Lab3实验文档

软件13 杨楠

# 操作环境

VMware Workstation Pro: Ubuntu 22.04.2 LTS

# Speed up system calls

修改kernel/proc.h文件，在struct proc中添加一个struct usyscall指针。

struct usyscall \*usyscallpage;

修改kernel/proc.c文件。

修改了proc\_pagetable(struct proc \*p)，添加了以下内容。同样是仿照其他PTE写法。这里权限是PTE\_R | PTE\_U。

if(mappages(pagetable, USYSCALL, PGSIZE,

(uint64)(p->usyscallpage), PTE\_R | PTE\_U) < 0){

uvmfree(pagetable, 0);

return 0;

}

修改allocproc()函数，添加以下内容。仿照了其他分配空间的写法。保存当前进程的pid。

if((p->usyscallpage = (struct usyscall \*)kalloc()) == 0){

freeproc(p);

release(&p->lock);

return 0;

}

p->usyscallpage->pid = p->pid;

在freeproc()函数中添加以下内容，释放该页。

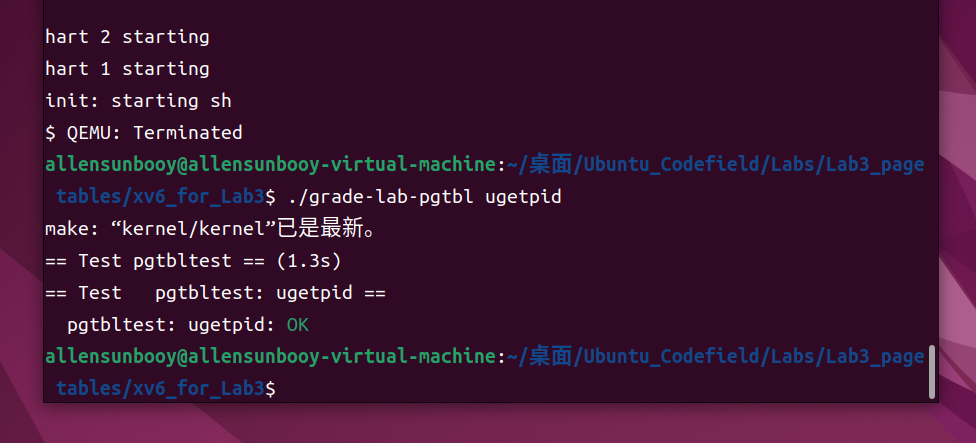
if(p->usyscallpage)

kfree((void \*)p->usyscallpage);

p->usyscallpage = 0;

在proc\_freepagetable()函数中添加以下内容，对该页取消PTE。

uvmunmap(pagetable, USYSCALL, 1, 0);



问题：

Which other xv6 system call(s) could be made faster using this shared page? Explain how.

比如fork()可以，通过在usyscall结构体中添加一个parent，供child在使用的时候不需要切到内核态。

# Print a page table

根据提示，要在kernel/vm.c中，实现vmprint()函数。

仿照了freewalk()函数的思路，使用了递归的方式。具体代码如下。

int deep = 0;

void

vmprint(pagetable\_t pagetable)

{

if(deep == 0)

printf("page table %p\n", (uint64)pagetable);

for(int i = 0; i < 512; i++){

pte\_t pte = pagetable[i];

if(pte & PTE\_V){

for(int j = 0; j <= deep; j++)

printf("..");

printf("%d: pte %p pa %p\n", i, (uint64)pte, (uint64)PTE2PA(pte));

}

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

deep++;

uint64 child = PTE2PA(pte);

vmprint((pagetable\_t)child);

deep--;

}

}

}

添加了变量deep表示页面深度。

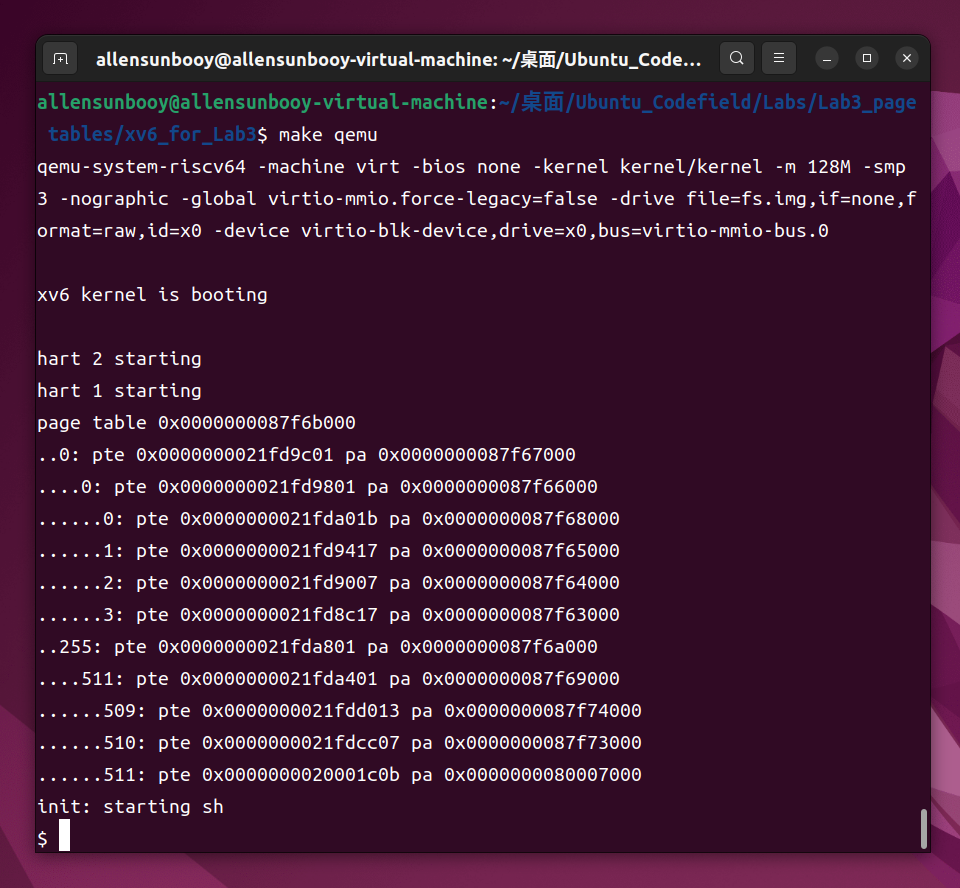
在kernel/defs.h中添加该函数的声明。

void vmprint(pagetable\_t);

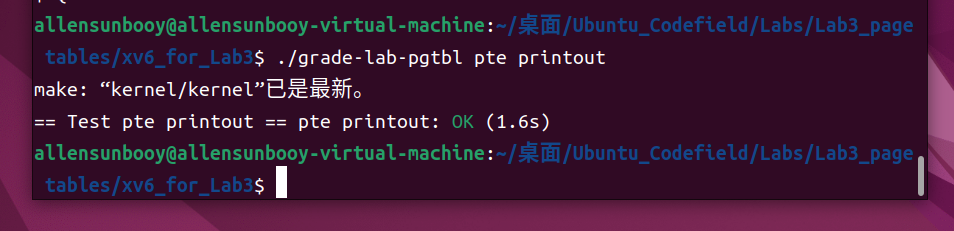
按题目要求在kernel/exec.c中，在return argc;前面添加：

if(p->pid==1)

vmprint(p->pagetable);



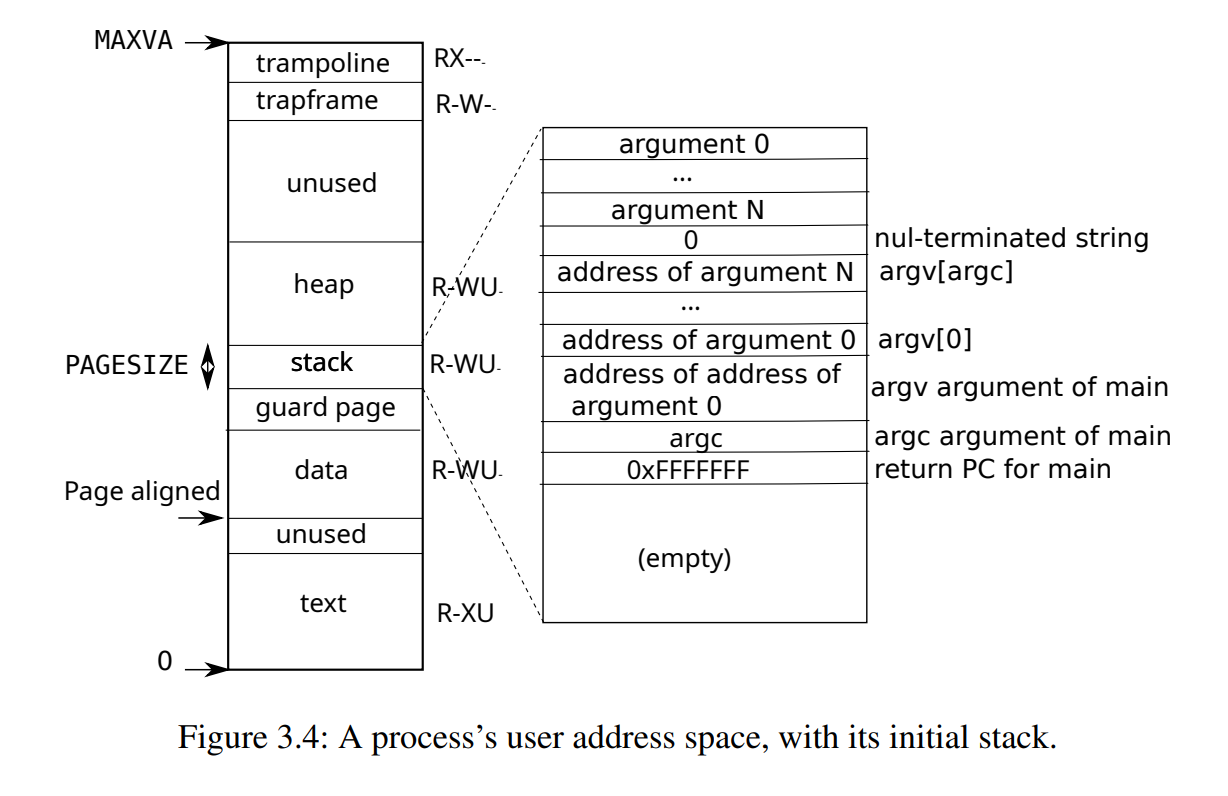
可以看到，能够按要求输出页表。



问题：

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? What does the third to last page contain?

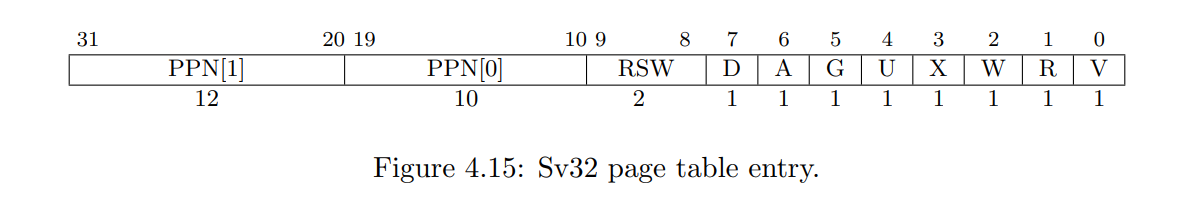
参考了xv6教材如下。



由图，page 0是data和text，page 2是stack，在用户态时，不能读写page 1，page 3到最后一页是heap, trapframe, trampoline

# Detect which pages have been accessed

在kernel/riscv.h中添加PTE\_A的定义。参考了RISC-V privileged architecture manual资料。



图中的A就是accessed bit，仿照其他PTE的定义，添加如下。

#define PTE\_A (1L << 6)

在kernel/sysproc.c中，实现sys\_pgaccess()函数。

int

sys\_pgaccess(void)

{

// lab pgtbl: your code here.

uint64 va;

int num;

uint64 abitsaddr;

argaddr(0, &va);

argint(1, &num);

argaddr(2, &abitsaddr);

// 设置扫描页面上限

if(num > 64)

{

panic("too much page");

return -1;

}

uint64 maskbits = 0;

struct proc \*proc = myproc();

for(int i = 0; i < num; i++){

pte\_t \*pte = walk(proc->pagetable, va + i \* PGSIZE, 0);

if(pte == 0)

panic("page not exist");

if(PTE\_FLAGS(\*pte) & PTE\_A)

maskbits = maskbits | (1L << i);

// 将PTI\_A位清零

\*pte = ((\*pte & PTE\_A) ^ \*pte) ^ 0;

}

if(copyout(proc->pagetable, abitsaddr, (char \*)&maskbits, sizeof(maskbits)) < 0)

return -1;

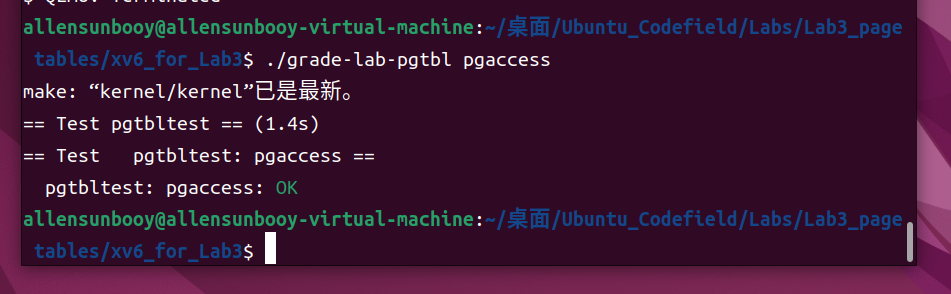
return 0;

}

按照实验提示，使用walk()函数，用于找到虚拟地址对应的PTE的地址。

设置了页面num的上限。

经测试，可以通过。



# 备注

提交的xv6\_for\_Lab3文件夹，已经过make clean处理。

