**Lab 3**

**Part A: Speed up system calls**

在proc.h中proc结构部分添加usyscall结构体指针\*usyscall。先在allocproc()中模仿trapframe为usyscall分配空间，如果分配失败则释放进程p以及锁，并直接返回0。分配完后根据提示将当前进程的pid存储至usyscall结构体指针中的pid部分。代码如下：

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

**freeproc**(p);

**release**(&p->lock);

return 0;

}

p->usyscall->pid = p->pid;

在freeproc()部分同样模仿trapframe释放usyscall，代码如下。

if(p->usyscall)

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

p->usyscall = 0;

随后，在proc\_pagetable()部分完成页到USYSCALL的映射。模仿先前TRAMPOLINE与TRAPFRAME部分，运用mappage()函数实现映射，并对返回值做判断：若返回值小于零，表示映射失败，因此需要通过uvmunmap()函数解除先前定义的映射关系，释放页表，并立即返回0。其中，mappage()函数的最后一个int变量值设置为PTE\_R ｜ PTE\_U，表示允许读，以及允许用户访问。完整代码如下：

if(**mappages**(pagetable, USYSCALL, **PGSIZE**, (**uint64**)(p->usyscall), **PTE\_R** | **PTE\_U**) < 0){

**uvmunmap**(pagetable, **TRAMPOLINE**, 1, 0);

**uvmunmap**(pagetable, **TRAPFRAME**, 1, 0);

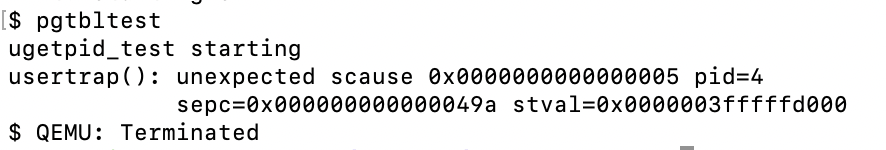
**uvmfree**(pagetable, 0);

return 0;

}

另外，在紧随其后的proc\_freepagetable()部分同样模仿TRAMPOLINE与TRAPFRAME解除USYSCALL映射。

完成以上全部工作后尝试运行，发现xv6报出以下错误信息：



检查代码后发现分配空间时usyscall的分配发生在页表的分配之后，因此调用proc\_pagetable()函数时无法实施USYSCALL的映射。修改后可正常运行得到结果，

……

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

**freeproc**(p);

**release**(&p->lock);

return 0;

}

p->usyscall->pid = p->pid;

*// An empty user page table.*

p->pagetable = **proc\_pagetable**(p);

……

**Part B: Print a page table**

根据提示在exec.c中加入相应语句。vmprint部分采取深度优先遍历的方式。对每个page table entry，用pte & PTE\_V判断该页表项有效位是否为1，若有效则通过PTE2PA宏来获取该页表项指向的物理地址，并按照提示规则进行打印。随后，由于指向下一级页表的页表项在读、写以及可执行位上均为零，因此可以通过pte & (PTE\_R | PTE\_W | PTE\_X)进行判断；若该页表项指向下一级页表，则将等级值减一并递归调用函数vmpwalk()。完成vmpwalk的编写后，在打印部分主函数vmprint中调用vmpwalk，并将初始等级设置为2。代码如下。

void **vmpwalk**(**pagetable\_t** pagetable, int level){

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

**pte\_t** pte = pagetable[i];

if (pte & **PTE\_V**){

**pagetable\_t** p\_addr = (**pagetable\_t**)**PTE2PA**(pte);

switch (level){

case 2: **printf**(".."); break;

case 1: **printf**(".. .."); break;

case 0: **printf**(".. .. .."); break;

}

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

if ((pte & (**PTE\_R** | **PTE\_W** | **PTE\_X**)) == 0) **vmpwalk**(p\_addr, level-1);

}

}

}

void **vmprint**(**pagetable\_t** pagetable){

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

**vmpwalk**(pagetable, 2);

}

**Part C: Detect which pages have been accessed**

首先在sysproc.c中的sys\_pgaccess()记录用户态传入的参数。其中，0号、1号、2号寄存器中分别存储初始地址、页数和结果存储地址。对于地址，用argaddr()函数获得参数；对于页数一类的整型变量则使用argint()。获得参数后，在该函数内调用pgaccess()并返回pgaccess的返回值。另外，在riscv.h部分需添加PTE\_A的宏定义，定义值根据xv6-book设置为1L<<6。这意味着每个页表项的第六位表示该页是否被访问过。代码如下：

#ifdef **LAB\_PGTBL**

int

**sys\_pgaccess**(void)

{

*// lab pgtbl: your code here.*

uint64 inipg\_addr;

int pg\_num;

uint64 store\_addr;

**argaddr**(0, &inipg\_addr);

**argint**(1, &pg\_num);

**argaddr**(2, &store\_addr);

return **pgaccess**(inipg\_addr, pg\_num, store\_addr);

}

#endif

随后，在proc.c末尾定义函数pgaccess()，利用传入参数中初始地址与页数对所有需要检查的页进行遍历，在每次遍历中通过当前页数与初始地址计算得到当前虚拟地址，并用walk()函数获得相应的页表项地址。获得页表项后通过(\*pte) & PTE\_A检查该页的访问情况，若被访问过则将变量bitmask的对应位的值改为一，并将该页的访问情况重置为零。最后，通过copyout()将结果交还给用户态，并返回copyout的返回值。代码如下。

**uint64**

**pgaccess**(**uint64** inipg\_addr, int pg\_num, **uint64** store\_addr){

int max\_num = 64;

struct **proc** \*p = **myproc**();

**pagetable\_t** pgtable = p->pagetable;

int bitmask = 0;

if (pg\_num > max\_num) return -1;

for (int i = 0; i < pg\_num; i++){

**uint64** virtual\_addr = inipg\_addr + i \* **PGSIZE**;

**pte\_t** \*pte = **walk**(pgtable, virtual\_addr, 0);

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

bitmask |= (1 << i);

(\*pte) &= ~**PTE\_A**;

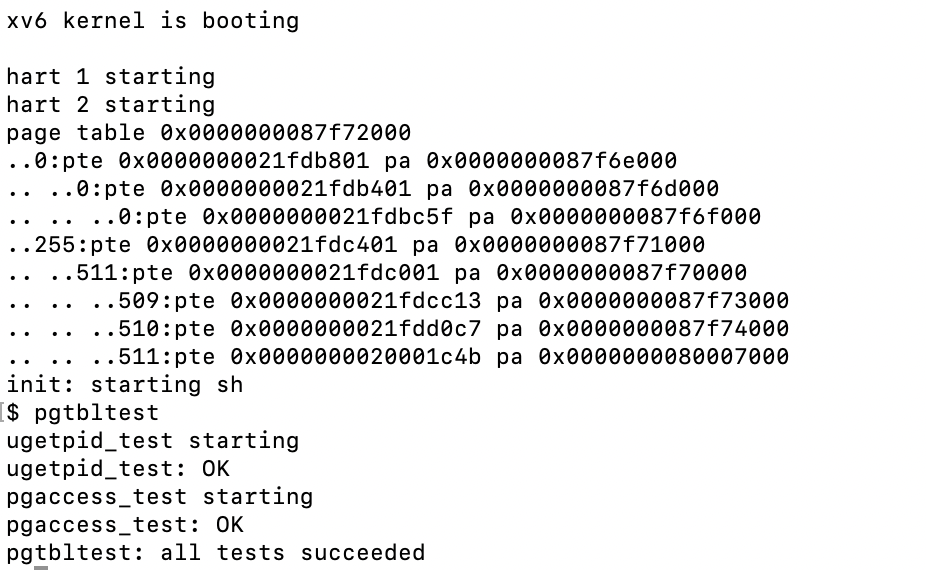
}

}

return **copyout**(pgtable, store\_addr, (char \*)&bitmask, sizeof(bitmask));

}

三个实验的最终结果如下：



**Questions**

1. **在 Part A 加速系统调用部分，除了getpid()系统调用函数，你还能想到哪些系统调用函数可以如此加速?**

Part A通过paging共享用户态和内核态的数据，因此凡是需要在用户态、内核态切换来获得数据的系统调用，亦即在系统调用函数中使用myproc()者均可以通过这种方式加速，如sys\_sbrk等。

1. **虚拟内存有什么用处？**

虚拟内存的理念之一在于让用户可见的地址空间趋近于无限：通过虚拟地址到物理地址到映射关系以及替换规则的设置，虚拟内存可以让用户在同样有限的物理地址下使用远大与其的虚拟地址空间。另外，虚拟内存还能够解决用户空间必须连续分配的问题。运用虚拟内存，用户使用的地址逻辑上连续，但在实际的物理空间中可能散落在天涯海角。

1. **为什么现代操作系统采用多级页表？**

单级页表容易造成页表过大，占用过多内存空间的问题。以32位地址空间为例，假设页大小为16KB，则页内偏移为14位，页帧号为18位，故而总共需要218个页表项。如果每个页表项为4Byte，则页表总共需占用220Byte，即1MB空间。若采用多级页表，假设外页号为8位，内页号为10位，则总共占用空间为210 + 212 Byte，即5KB，其远小于单级页表所占用的内存空间。

1. **简述Part C的detect流程。**

进入qemu后，在用户态下运行pgtbltest，运行至pgaccess\_test部分将切换至内核态并调用系统函数pgaccess。内核态首先经由sysproc.c将用户输入参数所在寄存器中的数值传入相应变量中并调用proc.c文件中定义的系统函数pgaccess。pgaccess通过循环遍历所有需要检查的页，通过当前页数及初始地址计算出当前虚拟地址并由此得到相应的页表项，随后通过PTE\_A检查该页表项是否被访问过，若被访问过，则将bitmask的对应位置为一，同时通过取否运算清空该页表项的访问状态。最后，通过copyout将结果交还用户态。

**Thoughts**

该实验分别实现了系统调用加速、页表打印以及页表项访问状态检查，其中包含页表映射、页表项信息查询等功能，加深了对于页表概念以及虚拟地址、物理地址间关系的理解。