Lab3

20307130340 杨孟颖 计算机

一、实验内容

1、 实现从用户态获取进程 pid 的函数 ugetpid，对获取 pid 进行加速。

2、 创建打印页表信息的函数 vmprint，实现以特定方式打印页表的 pte 和 pa。

3、 创建获取被访问页面的系统调用函数。需要第一个要检查的用户页面的起始虚拟地址并将将一个用户地址发送到一个缓冲区以存储结果进入位掩码(一种每页使用一个位的数据结构，并且第一页的位置对应于最低有效位)。

二、实验原理

使用到了数据类型 pagetable_t。它是 uint64*类型，指向存放 RISC-V 根页表的一页，它可以是内核的根页表，也可以是用户进程的根页表。

首先增加页表 USYSCALL

```
#define TRAMPOLINE (MAXVA - PGSIZE)
#define TRAPFRAME (TRAMPOLINE - PGSIZE)
#define USYSCALL (TRAPFRAME - PGSIZE)
```

USYSCALL 的位置

以下是从 xv6 book 获取的需要使用到的信号量的含义。

PTE_V 指示 PTE 是否存在/有效。如果不存在，尝试引用该页时就会引发一个缺页错误异常。

PTE_R 指示这一页物理帧是否能被读。

PTE_W 指示这一页物理帧是否能被写。

PTE_X 指示这一页物理帧是否能被 CPU 看待并转换成指令来执行。

PTE_U 指示这一页物理帧在用户模式下是否能访问。

Ugetpid 函数如下：

```
int
ugetpid(void)
{
  struct usyscall *u = (struct usyscall *)USYSCALL;
  return u->pid;
}
```

可以看出(struct usyscall *)USYSCALL 将内存页强制转换为结构体，而根据定义，USYSCALL 可以直接在用户态访问，所以访问时不需要进行用户态和 kernel 之间的转换。

在 proc.h 中添加：struct usyscall * usyscall;

在 proc.c 中确定映射关系的阶段 proc_pagetable(struct proc *p)函数中根据注释 map the trampoline code (for system call return)，在此函数中根据 TRAMPOLINE 映射添加 USYSCALL 的相应代码

```
  if(mappages(pagetable, TRAMPOLINE, PGSIZE,
              (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }
```

```
  // map the trapframe page just below the trampoline page, for
  // trampoline.S.
  if(mappages(pagetable, TRAPFRAME, PGSIZE,
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }
```

分配空间和确定映射关系：在 allocproc(void)函数中根据 trapframe page 的分配方式为 usyscall page 分配空间。含义是：若 kalloc 执行失败则释放指针 proc *p，并将&p->lock 释放。

```
// Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  // Allocate a usyscall page.
  if((p->usyscall = (struct usyscall *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  p->usyscall->pid = p->pid;
```

freeproc 函数释放：同样参照释放 trapframe 释放。

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

在 proc_freepagetable 函数中添加 uvmunmap(pagetable, USYSCALL, 1, 0);取消映射。

```
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, sz);
}
```

实验结果：

```
hart 2 starting
hart 1 starting
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

实验二

根据提示阅读 freewalk()函数部分，发现它使用了 DFS 算法遍历节点，因此在 vmprint()函数里也使用这一算法。

```
void _vmprint(pagetable_t pagetable, int depth){
    if(depth>2) return;
    for(int i = 0; i < 512; i++){
      pte_t pte = pagetable[i];
      if(pte & PTE_V){
        for(int j = 0;j<depth;j++) printf(".. ");
        printf("%d:pte %p pa %p\n",i,pte,PTE2PA(pte));
        uint64 child = PTE2PA(pte);
        _vmprint((pagetable_t)child, depth+1);
      }
    }
}
void vmprint(pagetable_t pagetable){
  printf("page table %p\n",pagetable);
  _vmprint(pagetable,0);
```

```
}
```

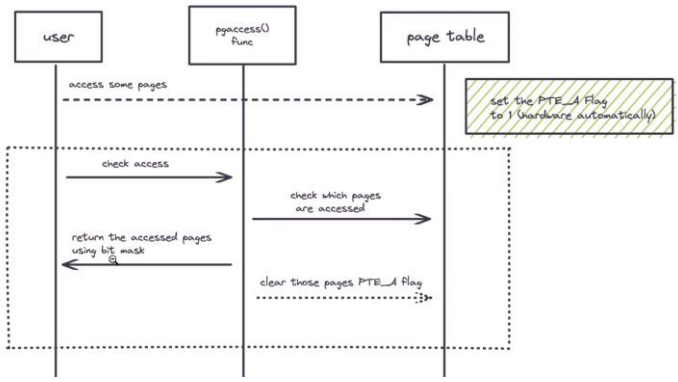
实验结果：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6b000
0:pte 0x0000000021fd9c01 pa 0x0000000087f67000
.. 0:pte 0x0000000021fd9801 pa 0x0000000087f66000
.. .. 0:pte 0x0000000021fda01b pa 0x0000000087f68000
.. .. 1:pte 0x0000000021fd9417 pa 0x0000000087f65000
.. .. 2:pte 0x0000000021fd9007 pa 0x0000000087f64000
.. .. 3:pte 0x0000000021fd8c17 pa 0x0000000087f63000
255:pte 0x0000000021fda801 pa 0x0000000087f6a000
.. 511:pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. 509:pte 0x0000000021fdcc13 pa 0x0000000087f73000
.. .. 510:pte 0x0000000021fdd007 pa 0x0000000087f74000
.. .. 511:pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

实验三

实验示意图如下：



根据 xv6 book 中的描述有下图的地址条件，从中得知 PTE_A 为第 7 位，参照其余有 PTE_A (1L << 6)。

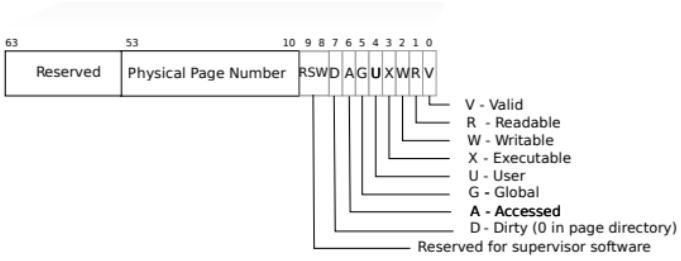


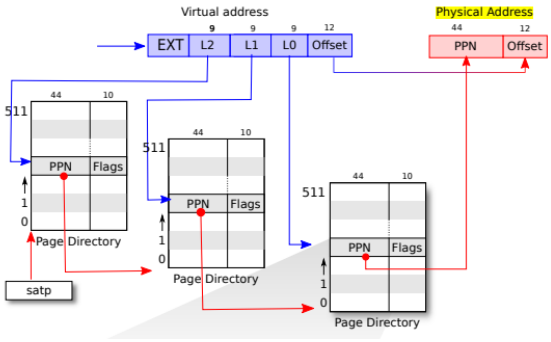
Figure 3.2: RISC-V address translation details.

在 riscv.h 中添加：

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
```

```
#define PTE_U (1L << 4) // user can access
#define PTE_A (1L << 6)
```

根据提示，阅读 walk()函数。walk()函数实现的是地址映射的功能。实现的功能如下图：使用 3 次映射将虚拟地址映射到物理地址中。




```
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

在 sys_proc.c 中添加

```
int
sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  int len;
  uint64 addr;
  int bitmask;
  argaddr(0,&addr);
  argint(1,&len);
  argint(2,&bitmask);
```

```
  struct proc *p = myproc();
  int res = 0x0;
  if(len>32||len<0) return -1;
  for(int i = 0;i<len;i++){
    int va = addr + i *PGSIZE;
    int abit = vmpgaccess(p->pagetable,va);
    res = res|abit<<i;
  }
  if(copyout(p->pagetable, bitmask, (char*)&res, sizeof(res)) < 0)
      return -1;
  return 0;
}
```

在 defs.h 中声明

```
int vmpgaccess(pagetable_t pagetable, uint64 va);
```

在 vm.c 中添加函数：

```
int vmpgaccess(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  if(va >= MAXVA)
    return 0;
  pte = walk(pagetable,va,0);
  if(pte==0) return 0;
  if((*pte & PTE_A)!=0){
    *pte = *pte & (~PTE_A);
    return 1;
  }
  return 0;
}
```

实验结果

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f6b000
0:pte 0x0000000021fd9c01 pa 0x0000000087f67000
.. 0:pte 0x0000000021fd9801 pa 0x0000000087f66000
.. .. 0:pte 0x0000000021fda01b pa 0x0000000087f68000
.. .. 1:pte 0x0000000021fd9417 pa 0x0000000087f65000
.. .. 2:pte 0x0000000021fd9007 pa 0x0000000087f64000
.. .. 3:pte 0x0000000021fd8c17 pa 0x0000000087f63000
255:pte 0x0000000021fda801 pa 0x0000000087f6a000
.. 511:pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. 509:pte 0x0000000021fdcc13 pa 0x0000000087f73000
.. .. 510:pte 0x0000000021fdd007 pa 0x0000000087f74000
.. .. 511:pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

问题：

（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？

Lab2 中获得系统调用的掩码 mask 的调用和跟踪系统调用的 trace。

（2） 虚拟内存有什么用处？

解决内存不足和内存碎片化的问题。虚拟内存通过映射的方式将程序中的虚拟地址映射到物理地址中，由于不同的程序的映射不同，所以对于每一个程序都认为内存是无限的。

（3） 为什么现代操作系统采用多级页表？

节省页表内存。使用一级页表，需要连续的内存空间来存放所有的页表项。多级页表通过只为进程实际使用的那些虚拟地址内存区请求页表来减少内存使用量。

（4） Detect 的过程

需要检查的是第一个页表的虚拟地址和以后的 n 个页表有没有被 access，然后将结果放在 bitmask 中。

首先通过：argaddr(0,&addr); argint(1,&len);

argint(2,&bitmask);获取页的数值情况，接受系统参数。

然后定义指向当前进程的 struct proc *p = myproc()，用来将后面得到的信息存入。

Walk 函数的作用是给出页表的虚拟地址可以得到页表的物理地址。且如果页表不合理，会分配空间新建页表。如果页表页合法且 PTE_A 合法，则将计算结果记录在相应的位上。

Copy 函数将信息从内核态拷贝到用户态。

Pgaccess 的作用是检验 PTE_A 是否为 0，并将 PTE_A 复位为 0.

过程为：先计算出页表的虚拟地址，然后通过 pgaccess 函数计算当前页表的 accessbit 有没有使用。然后将计算的结果记录在 res 中。

实验中出现的问题：

没有在 exec.c 文件中声明函数

void vmprint(pagetable_t pagetable);

```
yangmy@ubuntu:~/xv6-lab3/xv6-labs-2022$ make qemu
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTBL
-DLAB_PGTBL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-st
ack-protector -fno-pie -no-pie   -c -o kernel/vm.o kernel/vm.c
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_PGTBL
-DLAB_PGTBL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-st
ack-protector -fno-pie -no-pie   -c -o kernel/exec.o kernel/exec.c
kernel/exec.c: In function ‘exec’:
kernel/exec.c:130:17: error: implicit declaration of function ‘vmprint’ [-Werror=implicit-f
unction-declaration]
  130 |   if(p->pid==1) vmprint(p->pagetable);
      |                 ^~~~~~~
cc1: all warnings being treated as errors
make: *** [Makefile:130: kernel/exec.o] Error 1
```