Mit-Lab3: Page tables

20307130350 陈丹纯 信息安全 2022/11/08

Web

Lab: page tables (mit.edu)

Speed up system calls

实验目标

1. speed up `getpid()` syscall by sharing data in a read-only region between userpace and the kernel.
2. Further learn how to insert mappings into a page table.

实验内容

> 当一个进程被创建的时候，映射一个只读页到虚拟地址`USYSCALL`（定义在`memlayout.h`中）。
> 在该页的开始存放一个结构体，初始化这个结构体，结构体的只有一个成员，初始化为当前进程的`pid`。

实验验收：运行`pgtbltest`, 通过`ugetpid` test.

实验思路

1. check `ugetpid_test()` in `pgtbltest.c`. We can see that the key is `getpid() == ugetpid()`.

```
28 void
29 ugetpid_test()
30 {
31   int i;
32
33   printf("ugetpid_test starting\n");
34   testname = "ugetpid_test";
35
36   for (i = 0; i < 64; i++) {
37     int ret = fork();
38     if (ret != 0) {
39       wait(&ret);
40       if (ret != 0)
41         exit(1);
42       continue;
43     }
44     if (getpid() != ugetpid())
45       err("missmatched PID");
46     exit(0);
47   }
48   printf("ugetpid_test: OK\n");
49 }
```

2. check `ugetpid()` in `user/slib.c`. 可见，该进程的地址USYSCALL被强转为`usyscall*` 类型,返回其`pid`.

```
154 #ifdef LAB_PGTBL
155 int
156 ugetpid(void)
157 {
158   struct usyscall *u = (struct usyscall *)USYSCALL;
159   return u->pid;
160 }
161 #endif
```

3. check `USYCALL` page in `kernel/memlayout.h`.

```
72 #define TRAPFRAME (TRAMPOLINE - PGSIZE)
73 #ifdef LAB_PGTBL
74 #define USYSCALL (TRAPFRAME - PGSIZE)
75
76 struct usyscall {
77   int pid;  // Process ID
78 };
79 #endif
```

综合123，可以知道需要让`getpid()`的pid等于USYSCALL的pid。

并且根据提示，在`kernel/proc.c`中的`proc_pagetable`映射`USYSCALL` page.

4. check `proc_pagetable`

```
// map the trampoline code (for system call return)
// at the highest user virtual address.
// only the supervisor uses it, on the way
// to/from user space, so not PTE_U.
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;
```

观察到两个映射页面的操作，即`mappages`。

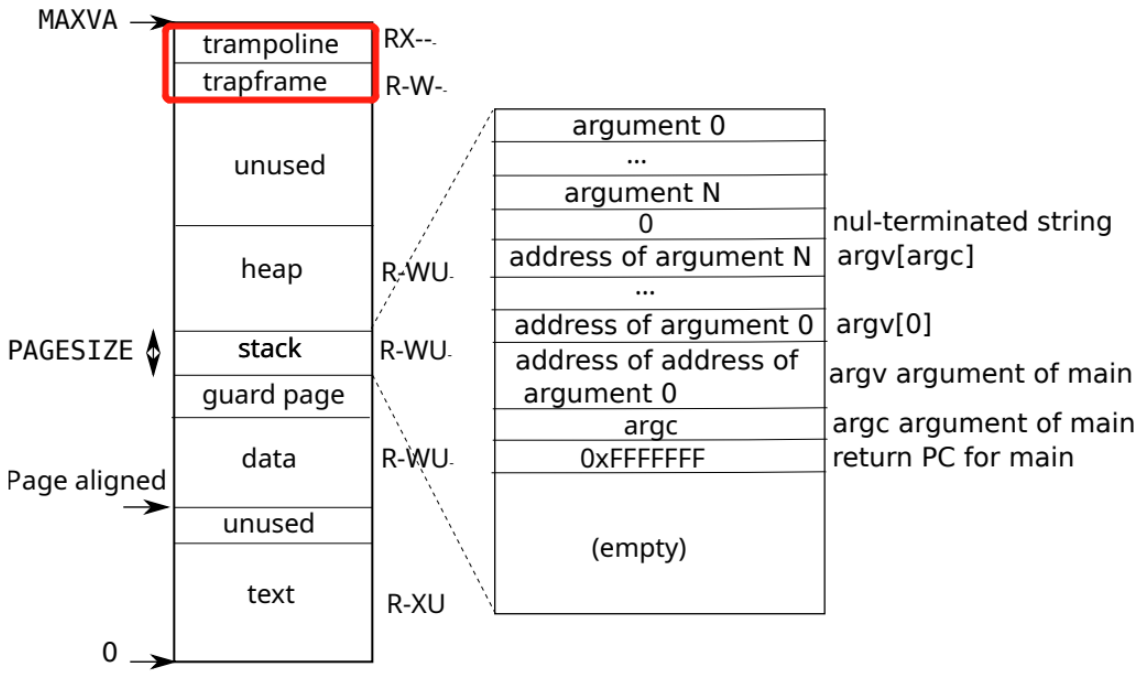


Figure 3.4: A process's user address space, with its initial stack.

而由3和`xv6 book`可知，只需要仿照`trapframe`创建和映射`usyscall`即可。

实验步骤

1. define `usyscall*` in struct `proc` in `proc.h`.

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S

  struct usyscall *usyscall;   // data page for usyscall (add for lab3)

  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

2. allocate and initialize(by assigning the proc's pid to its own) the `usyscall` page in `allocproc()`.

```c
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->pid = allocpid();
  p->state = USED;

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

  // Allocate a usyscall page. (add for lab3)
  if((p->usyscall = (struct usyscall *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  p->usyscall->pid = p->pid; //分配pid
```

6. mapping the `usyscall` in `proc_pagetable()`.

```c
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;
}

// add for lab3
// map the usyscall page just below the trampframe page, for
// read-only usyscall
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;
}
```

7. free the page in `freeproc()` and `proc_freepagetable()`.

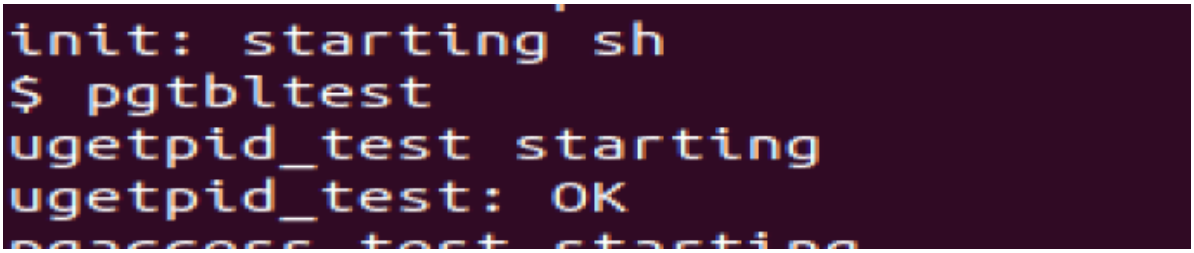
```
// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;

  //add for lab3
  if(p->usyscall)
    kfree((void*)p->usyscall);
  p->usyscall = 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;
}

// Free a process's page table, and free the
// physical memory it refers to.
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmunmap(pagetable, USYSCALL, 1, 0); //Add for lab3 to free usyscall
  uvmfree(pagetable, sz);
}
```

实验结果



Print a page table

实验目标

To help you visualize RISC-V page tables, and perhaps to aid future debugging, your second task is to write a function that prints the contents of a page table.

实验内容

1. Define a function called `vmprint()` in `kernel/vm.c`.
2. It should take a `pagetable_t` argument, and print that pagetable in the format described below.
3. Insert `if(p->pid==1) vmprint(p->pagetable)` in exec.c just before the `return argc`, to print the first process's page table.
4. You receive full credit for this part of the lab if you pass the `pte printout` test of `make grade`.

实验思路

According to the `freewalk` in `vm.c`, we can imitate it to write the `vmprint()`.

```
// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void
freewalk(pagetable_t pagetable)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      freewalk((pagetable_t)child);
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      panic("freewalk: leaf");
    }
  }
  kfree((void*)pagetable);
}
```

由要求的输出结果，可以确定该函数需要使用三层循环+递归。

```
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
```

实验步骤

1. implement `vmprint()` in `kernel/vm.c`.

```
void
vmprint(pagetable_t pagetable)
{
  printf("page table %p\n",pagetable);
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t top_pte = pagetable[i];
    if(top_pte & PTE_V){
      pagetable_t mid_pagetable = (pagetable_t) PTE2PA(top_pte);
      printf(".. %d: pte %p pa %p\n",i,top_pte,mid_pagetable);
      for(int j = 0; j < 512; j++){
        pte_t mid_pte = mid_pagetable[j];
        if(mid_pte & PTE_V){
          pagetable_t leaf_pagetable = (pagetable_t) PTE2PA(mid_pte);
          printf(".. .. %d: pte %p pa %p\n",j,mid_pte,leaf_pagetable);
          for(int k = 0; k < 512; k++){
            pte_t leaf_pte = leaf_pagetable[k];
            if(leaf_pte & PTE_V){
              printf(".. .. .. %d: pte %p pa %p\n",k,leaf_pte,PTE2PA(leaf_pte));
              }
            }
          }
        }
      }
    }
}
```

note:

- 因为输出结果pagetable和pte的输出不同。所以不方便直接用递归（也可再定义一个printpte的函数实现递归），但是只有三层，所以直接用三层循环完成。

- Use `%p` in your printf calls to print out full 64-bit hex PTEs and addresses as shown in the example.

2. Define the prototype for `vmprint` in kernel/defs.h so as to call it from `exec.c`.

```
157
158 // vm.c
159 void               kvminit(void);
160 void               kvminithart(void);
161 void               kvmmap(pagetable_t, uint64, uint64, uint64, int);
162 int                mappages(pagetable_t, uint64, uint64, uint64, int);
163 pagetable_t        uvmcreate(void);
164 void               uvmfirst(pagetable_t, uchar *, uint);
165 uint64             uvmalloc(pagetable_t, uint64, uint64, int);
166 uint64             uvmdealloc(pagetable_t, uint64, uint64);
167 int                uvmcopy(pagetable_t, pagetable_t, uint64);
168 void               uvmfree(pagetable_t, uint64);
169 void               uvmunmap(pagetable_t, uint64, uint64, int);
170 void               uvmclear(pagetable_t, uint64);
171 pte_t *            walk(pagetable_t, uint64, int);
172 uint64             walkaddr(pagetable_t, uint64);
173 int                copyout(pagetable_t, uint64, char *, uint64);
174 int                copyin(pagetable_t, char *, uint64, uint64);
175 int                copyinstr(pagetable_t, char *, uint64, uint64);
176 void               vmprint(pagetable_t);
```

3. Insert `if(p->pid==1) vmprint(p->pagetable)` in exec.c just before the `return argc`, to print the first process's page table.

```
p->trapframe->sp = sp; // initial stack pointer
proc_freepagetable(oldpagetable, oldsz);
//add for lab3.2
if(p->pid==1)
      vmprint(p->pagetable);
return argc; // this ends up in a0, the first argument to main(argc, argv)
```

实验结果

```
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
```

Detect which pages have been accessed

实验内容

1. implement `pgaccess()`, a system call that reports which pages have been accessed.
2. The system call takes three arguments. First, it takes the starting virtual address of the first user page to check. Second, it takes the number of pages to check. Finally, it takes a user address to a buffer to store the results into a bitmask (a datastructure that uses one bit per page and where the first page corresponds to the least significant bit).
3. You will receive full credit for this part of the lab if the `pgaccess` test case passes when running `pgtbltest`.

实验思路

check `pgaccess_test()` in `pgtbltest.c`.

```
void
pgaccess_test()
{
  char *buf;
  unsigned int abits;
  printf("pgaccess_test starting\n");
  testname = "pgaccess_test";
  buf = malloc(32 * PGSIZE);
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  buf[PGSIZE * 1] += 1;
  buf[PGSIZE * 2] += 1;
  buf[PGSIZE * 30] += 1;
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  if (abits != ((1 << 1) | (1 << 2) | (1 << 30)))
    err("incorrect access bits set");
  free(buf);
  printf("pgaccess_test: OK\n");
}
```

可以看出：pgaccess的用法：给buf分配了32页，用一个int类型变量abits存储被访问的page。

因此只需要按照提示实现 `sys_pgaccess()`。

实验步骤

implementing `sys_pgaccess()` in `kernel/sysproc.c`.

```
#ifdef LAB_PGTBL
int
sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  uint64 startaddr;
  int pagenum;
  uint64 maskaddr;
  argaddr(0,&startaddr);
  argint(1,&pagenum);
  argaddr(2,&maskaddr);

  pagetable_t pt = myproc()->pagetable;
  uint64 bit_mask=0;


  for(int i=0;i<pagenum;i++){
        pte_t *pte =  walk(pt, startaddr+i*PGSIZE,0);
        if(*pte & PTE_A){
                bit_mask = bit_mask | (1<<i);
                *pte = *pte & (~PTE_A);
                printf("%d bit_mask:%p\n",i,bit_mask);
        }
  }
  copyout(pt, maskaddr,(char*)&bit_mask,sizeof(uint64));
  return 0;
}
#endif
```

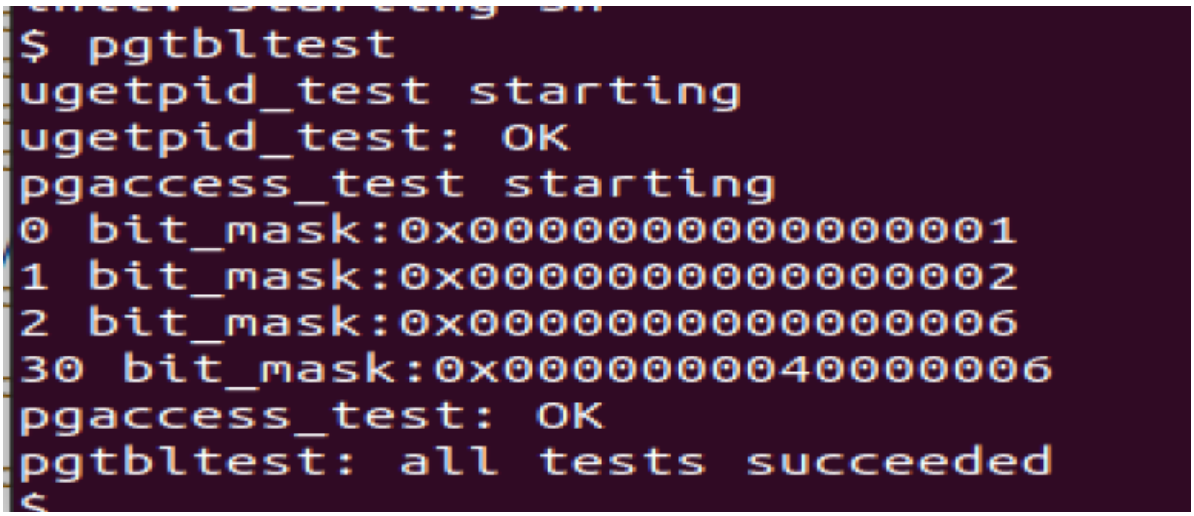
1. use `argint` and `argaddr` to parse 3 arguments from user space to kernel space.
2. 设置一个64位的bit_mask,来记录accessed page.
    - 根据pgtbltest中的使用，也可设置为32位。
3. 遍历pagetable,使用walk函数找到对应的PTE，如果PTE_A有效，则通过与`1<<i`来重置bit_mask.
    - bitmask (a datastructure that uses one bit per page and where **the first page corresponds to the least significant bit**).
4. 通过`*pte&(~PTE_A)`重置pte，clear `PTE_A` after checking if it is set.

- Otherwise, it won't be possible to determine if the page was accessed since the last time `pgaccess()` was called (i.e., the bit will be set forever).

5. define `PTE_A`, the access bit, in `kernel/riscv.h`. （根据risc-v manual）

```
40
41 #define PTE_V (1L << 0) // valid
42 #define PTE_R (1L << 1)
43 #define PTE_W (1L << 2)
44 #define PTE_X (1L << 3)
45 #define PTE_U (1L << 4) // user can access
46 #define PTE_A (1L << 6) //Add for lab3 (access bit)
47
```

实验结果



问题回答

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?

wait(); fork()

2. 虚拟内存有什么用处?

在用户和物理地址之间抽象出的一种概念，通过给用户制造一种无限内存的假象，便于用户使用内存。同时通过设置RWXV等bits，虚拟内存也实现了进程间的保护和isolation。

3. 为什么现代操作系统采用多级页表?

因为单级页表的page table所需要存储空间大，反而加重了os的负载，造成空间复杂度的浪费。此外，如果只是去掉没有用到的page导致页号不连续，则mapping过程无法直接通过索引查找，需要加载内存来检索页号，反而会增加时间复杂度。因此，通过多级页表，可以在降低空间复杂性的同时也不增大时间复杂度。

4. 简述 Part C 的 detect 流程。

- user-space：

call `pgaccess()` in `user/pgtbltest.c`->makefile invokes the perl scipt user/usys.pl to produce the actual true syscall stubs user\usys.S , which ecall to the kernel space.

- kernel-space:

`syscall.c` to save the caller's stack and call syscall(sys_pgacess) in sysproc.c , which transfer the argument from the user-space, and recode the accessed pages in the bit_mask, which is transferred to the userspace. Then return to syscall.c to run the system call.

After the user get the bit_mask, do the test.

实验问题及解决

1. part C的时候一直fail：

```
ugetpid_test: OK
pgaccess_test starting
pgtbltest: pgaccess_test failed: incorrect access bits set, pid=4
```

通过vmprint来查看是否有accessed page：

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
page table 0x0000000087f40000
.. 0: pte 0x0000000021fcf001 pa 0x0000000087f3c000
.. .. 0: pte 0x0000000021fcec01 pa 0x0000000087f3b000
.. .. .. 0: pte 0x0000000021fcf45b pa 0x0000000087f3d000
.. .. .. 1: pte 0x0000000021fce8d7 pa 0x0000000087f3a000
.. .. .. 2: pte 0x0000000021fce407 pa 0x0000000087f39000
.. .. .. 3: pte 0x0000000021fce0d7 pa 0x0000000087f38000
.. .. .. 4: pte 0x0000000021fdb8d7 pa 0x0000000087f6e000
.. .. .. 5: pte 0x0000000021fdc017 pa 0x0000000087f70000
.. .. .. 6: pte 0x0000000021fd7c17 pa 0x0000000087f5f000
.. .. .. 7: pte 0x0000000021fdbc17 pa 0x0000000087f6f000
.. .. .. 8: pte 0x0000000021fd0c17 pa 0x0000000087f43000
.. .. .. 9: pte 0x0000000021fd1817 pa 0x0000000087f46000
.. .. .. 10: pte 0x0000000021fd0817 pa 0x0000000087f42000
.. .. .. 11: pte 0x0000000021fd1017 pa 0x0000000087f44000
.. .. .. 12: pte 0x0000000021fd8017 pa 0x0000000087f60000
.. .. .. 13: pte 0x0000000021fd1417 pa 0x0000000087f45000
.. .. .. 14: pte 0x0000000021fd0417 pa 0x0000000087f41000
.. .. .. 15: pte 0x0000000021fd1c17 pa 0x0000000087f47000
.. .. .. 16: pte 0x0000000021fd2017 pa 0x0000000087f48000
.. .. .. 17: pte 0x0000000021fd2417 pa 0x0000000087f49000
.. .. .. 18: pte 0x0000000021fd2817 pa 0x0000000087f4a000
.. .. .. 19: pte 0x0000000021fd2c17 pa 0x0000000087f4b000
.. .. .. 20: pte 0x0000000021fd3017 pa 0x0000000087f4c000
.. .. .. 21: pte 0x0000000021fd3417 pa 0x0000000087f4d000
.. .. .. 22: pte 0x0000000021fd3817 pa 0x0000000087f4e000
.. .. .. 23: pte 0x0000000021fd3c17 pa 0x0000000087f4f000
.. .. .. 24: pte 0x0000000021fd4017 pa 0x0000000087f50000
.. .. .. 25: pte 0x0000000021fd4417 pa 0x0000000087f51000
.. .. .. 26: pte 0x0000000021fd4817 pa 0x0000000087f52000
.. .. .. 27: pte 0x0000000021fd4c17 pa 0x0000000087f53000
.. .. .. 28: pte 0x0000000021fdb417 pa 0x0000000087f6d000
.. .. .. 29: pte 0x0000000021fd8417 pa 0x0000000087f61000
.. .. .. 30: pte 0x0000000021fd8817 pa 0x0000000087f62000
.. .. .. 31: pte 0x0000000021fcdc17 pa 0x0000000087f37000
.. .. .. 32: pte 0x0000000021fcd817 pa 0x0000000087f36000
.. .. .. 33: pte 0x0000000021fcd417 pa 0x0000000087f35000
.. .. .. 34: pte 0x0000000021fcd017 pa 0x0000000087f34000
.. .. .. 35: pte 0x0000000021fccc17 pa 0x0000000087f33000
.. .. .. 36: pte 0x0000000021fcc817 pa 0x0000000087f32000
.. 255: pte 0x0000000021fcfc01 pa 0x0000000087f3f000
```

确实是有accessed page的。怀疑是bit_mask的问题，在判断条件前打印mask。

```
.. .. .. 511: pte 0x0000000020001c4b pa 0x0000000080007000
0 bit_mask:0x0000000000000000
1 bit_mask:0x0000000000000000
2 bit_mask:0x0000000000000000
3 bit_mask:0x0000000000000000
4 bit_mask:0x0000000000000000
5 bit_mask:0x0000000000000000
6 bit_mask:0x0000000000000000
7 bit_mask:0x0000000000000000
8 bit_mask:0x0000000000000000
9 bit_mask:0x0000000000000000
10 bit_mask:0x0000000000000000
11 bit_mask:0x0000000000000000
12 bit_mask:0x0000000000000000
13 bit_mask:0x0000000000000000
14 bit_mask:0x0000000000000000
15 bit_mask:0x0000000000000000
16 bit_mask:0x0000000000000000
17 bit_mask:0x0000000000000000
18 bit_mask:0x0000000000000000
19 bit_mask:0x0000000000000000
20 bit_mask:0x0000000000000000
21 bit_mask:0x0000000000000000
22 bit_mask:0x0000000000000000
23 bit_mask:0x0000000000000000
24 bit_mask:0x0000000000000000
25 bit_mask:0x0000000000000000
26 bit_mask:0x0000000000000000
27 bit_mask:0x0000000000000000
28 bit_mask:0x0000000000000000
29 bit_mask:0x0000000000000000
30 bit_mask:0x0000000000000000
31 bit_mask:0x0000000000000000
pgtbltest: pgaccess_test failed: incorrect access bits set, pid=3
$
```

发现bit_mask并没有改变，是因为没有进入循环，说明PTE_A有问题。

```
 83   argaddr(2,&maskaddr);
 84
 85   pagetable_t pt = myproc()->pagetable;
 86   uint64 bit_mask=0;
 87
 88   vmprint(pt);
 89   for(int i=0;i<pagenum;i++){
 90         pte_t *pte =  walk(pt, startaddr+i*PGSIZE,0);
 91         printf("%d bit_mask:%p\n",i,bit_mask);
 92         if(*pte & PTE_A){
 93                 bit_mask = bit_mask | (1<<i);
 94                 *pte = *pte & (~PTE_A);
 95                 printf("%d bit_mask:%p\n",i,bit_mask);
 96         }
 97   }
 98   copyout(pt, maskaddr,(char*)&bit_mask,sizeof(uint64));
 99   return 0;
100 }
101 #endif
```

一开始想当然设置的是5；因为是这些pte bits是硬件支持，通过查阅risc-v手册，发现该位固定为6；

```
345 #define PTE_U (1L << 4) // user can access
346 #define PTE_A (1L << 5) //Add for lab3 (access bit)
347
```

```
45 #define PTE_U (1L << 4) // user can access
46 #define PTE_A (1L << 6) //Add for lab3 (access bit)
```

修改之后，即detect成功。

```
== Test pgtbltest ==
$ make qemu-gdb
m(2.8s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.8s)
```

实验感想

1. 通过本实验part C，我将lab2所学的系统调用相关函数学以致用，进一步加深了对调用system call的过程的理解。
2. 通过阅读xv6 book和实战进一步了解到page的映射，创建，free等过程。对trapframe等page也有所了解。