

Lab3 Page Table

邹怡21307130422

part1 加速系统调用速度

这个实验的原理就是，将一些数据存放到一个只读的共享空间中，这个空间位于内核和用户之间。这样用户程序就不用陷入内核中，而是直接从这个只读的空间中获取数据，省去了一些系统开销，加速了一些系统调用。这次的任务是改进 `getpid()`。

- 在 `kernel/proc.h` `proc` 结构体中添加一项指针来保存页表地址
- 在 `kernel/proc.c` 的 `allocproc()` 中为其分配空间(`kalloc`)。并初始化其保存当前进程的pid
- 在 `kernel/proc.c` 的 `proc_pagetable()` 中将这个映射 (PTE) 写入 `pagetable` 中，权限是用户态可读
- 在 `kernel/proc.c` 的 `freeproc()` 中确保释放进程的时候，能够释放该共享页，将页表插入空闲页链表中
- 在 `kernel/proc.c` 的 `proc_freepagetable()` 中解除映射关系 1.先根据`pgtbltest.c`中的函数调用

```
if (getpid() != ugetpid())
```

再找到`ugetpid()`函数的调用：

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

并查看`usyscall`结构体组成，发现保存了pid

```
#define USYSCALL (TRAPFRAME - PGSIZE)

struct usyscall {
    int pid; // Process ID
};

#endif
```

当每一个进程被创建，映射一个只读的页在 `USYSCALL`（在 `memlayout.h` 定义的一个虚拟地址）处。存储一个 `struct usyscall`（定义在 `memlayout.h`）结构体在该页的开始处，并且初始化这个结构体来保存当前进程的 PID。这个 lab 中，`ugetpid()` 已经在用户空间给出，它将会使用 `USYSCALL` 这个映射。2. 故先在 `kernel/proc.h` 中的 `proc` 结构体中增加一个指针保存用户系统调用共享页面的地址：`struct usyscall *usyscallpage`; 3. 接着在 `kernel/proc.c` 的 `allocproc()` 中为其分配空间。并初始化其保存当前进程的 PID。

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

if((p->usyscallpage = (struct usyscall *)kalloc()) == 0)
{
    freeproc(p);
    release(&p->lock);
    return 0;
}
p->usyscallpage->pid = p->pid;
// An empty user page table.
p->pagetable = proc_pagetable(p);
if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
}
```

4.在kernel/proc.c 的proc_pagetable()中将映射写入pagetable, 这部分代码参考trapframe写入的部分, 并且将其设置为用户可读 参考代码:

```
// 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;  
}
```

依据上面代码分析mappages()函数的参数分别代表:

- pgdir: 这是一个指向页目录的指针, 它指向要执行映射操作的页目录。
- va: 虚拟地址 (Virtual Address), 是要映射的虚拟地址。
- sz: 映射的大小, 通常以页 (Page) 为单位, 通常是 PGSIZE, 即页面的大小。
- pa: 物理地址 (Physical Address), 是要映射到的物理地址。
- perm: 权限标志 (Permission Flags), 这是一个包含 PTE 标志的变量, 它定义了映射的权限, 例如读取、写入、用户态权限等。

故仿写上面参考代码, 同时要注意将上面映射好的都释放掉:

```
// 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->usyscallpage), PTE_R | PTE_U) < 0){  
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);  
    uvmunmap(pagetable, TRAPFRAME, 1, 0); //释放上面的映射好的  
    uvmfree(pagetable, 0);  
    return 0;  
}  
  
return pagetable;  
}
```

5.在kernel/proc.c中的freeproc()进行释放 参考代码:

```
if(p->trapframe)
    kfree((void*)p->trapframe);
p->trapframe = 0;
```

实现：

```
//new
if(p->usyscallpage)
    kfree((void*)p->usyscallpage);
p->usyscallpage = 0;
```

6.在proc_freepagetable()中增加语句解除映射关系

```
// physical memory / 4 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); //new
    uvmfree(pagetable, sz);
}
```

实验结果

```
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$ QEMU: Terminated
```

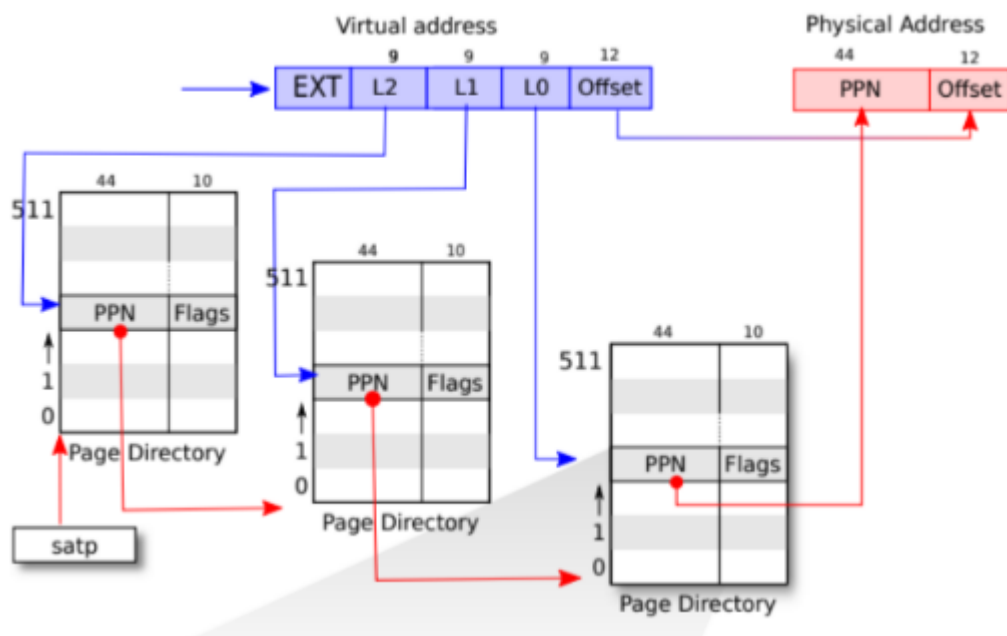
part2 打印页表

写一个函数来打印页表的内容。这个函数定义为 `vmprint()`。它应该接收一个 `pagetable_t` 类型的参数，并且按照下面的格式打印。在 `exec.c` 中的 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)`，用来打印第一个进程的页表。

可以将 `vmprint()` 实现到 `kernel/vm.c` 中。
使用在 `kernel/riscv.h` 文件末尾的宏定义。
函数 `freewalk` 的实现方法对本实验很有帮助。
将函数 `vmprint` 的声明放到 `kernel/defs.h` 中，以便可以在 `exec.c` 中调用它。
使用 `%p` 格式化打印64位十六进制的 PTEs 和 地址。

1.先查看`freewalk`函数的具体实现，该函数释放所有页表，是通过递归进行释放，其中`pte_t`是页表项数据类型，储存了页表的相关信息，子页表的信息可以通过PTE2PA函数进行获取

```
6 void
7 freewalk(pagetable_t pagetable)
8 {
9     // there are 2^9 = 512 PTEs in a page table.
10    for(int i = 0; i < 512; i++){
11        pte_t pte = pagetable[i];
12        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
13            // this PTE points to a lower-level page table.
14            uint64 child = PTE2PA(pte);
15            freewalk((pagetable_t)child);
16            pagetable[i] = 0;
17        } else if(pte & PTE_V){
18            panic("freewalk: leaf");
19        }
20    }
21    kfree((void*)pagetable);
22 }
```



2.由于页表只有三级，故可以使用迭代法三层循环进行打印,在第一层循环中获取页表，并进行判断，获取下一级页表以后进行打印并进入下一层迭代，如此进行3次

```
//new vmprint
void
vmprint(pagetable_t pagetable){
    printf("page table %p\n",pagetable);
    for(int i=0;i<512;i++){
        {
            pte_t pte = pagetable[i];
            if(pte&PTE_V)
            {
                uint64 pa2 = PTE2PA(pte); //pa2存储下一级页表
                printf("..%d: pte %p pa %p\n",i,pte,pa2);
                for(int j=0;j<512;j++){
                    {
                        pagetable_t pagetable1 = (pagetable_t)pa2; //重新转化成页表数据类型
                        pte_t pte=pagetable1[j];
                        if(pte&PTE_V)
                        {
                            uint64 pa1=PTE2PA(pte);
                            printf(".. ..%d: pte %p pa %p\n",j,pte,pa1);

                            for(int k=0;k<512;k++){
                                {
                                    pagetable_t pagetable0=(pagetable_t) pa1;
                                    pte_t pte=pagetable0[k];
                                    if(pte&PTE_V)
                                    {
                                        uint64 pa0=PTE2PA(pte);
                                        printf(".. .. ..%d: pte %p pa %p\n",k,pte,pa0);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
}  
}  
}
```

3.在exec.c中调用函数

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

4.在defs.h中进行声明

```
// vm.c  
void    kvminit(void);  
void    kvminithart(void);  
void    kvmmap(pagetable_t, uint64, uint64, uint64, int);  
int      mappages(pagetable_t, uint64, uint64, uint64, int);  
pagetable_t  uvmcreate(void);  
void      uvmfirst(pagetable_t, uchar *, uint);  
uint64    uvmalloc(pagetable_t, uint64, uint64, int);  
uint64    uvmdealloc(pagetable_t, uint64, uint64);  
int        uvmcopy(pagetable_t, pagetable_t, uint64);  
void        uvmfree(pagetable_t, uint64);  
void        uvmunmap(pagetable_t, uint64, uint64, int);  
void        uvmclear(pagetable_t, uint64);  
pte_t *     walk(pagetable_t, uint64, int);  
uint64      walkaddr(pagetable_t, uint64);  
int          copyout(pagetable_t, uint64, char *, uint64);  
int          copyin(pagetable_t, char *, uint64, uint64);  
int          copyinstr(pagetable_t, char *, uint64, uint64);  
void          vmprint(pagetable_t pagetable);
```

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 0x0000000021fdc013 pa 0x0000000087f70000
.. .. 510: pte 0x0000000021fdd007 pa 0x0000000087f74000
.. .. 511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ pgtbltest
```

```
= Test pte printout =
$ make qemu-gdb
pte printout: OK (0.8s)
```

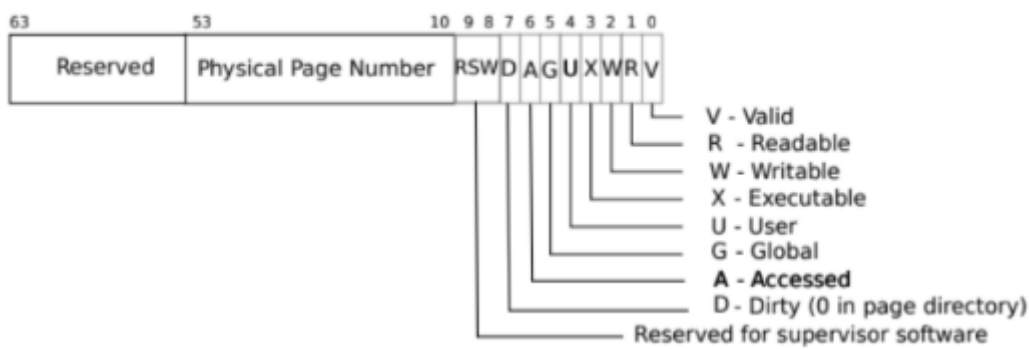
part3 检测哪些页被访问

实现 `pgaccess()`, 作用是报告哪些页被访问过。此系统调用接受三个参数, 第一: 被检查的第一个用户页的起始虚拟地址; 第二: 被检查页面的数量; 第三: 接收来自用户地址空间的一个 `buffer` 地址, 将结果以掩码

(bitmask) 的形式写入 1。先分析 `pgaccess_test`, 创建了一个 `buf`, 给他分配了 32 页, 然后修改其中三页, 看是否能返回正确的内容。

```
void
pgaccess_test()
{
    char *buf;
    unsigned int abits;
    printf("pgaccess_test starting\n");
    testname = "pgaccess_test";
    buf = malloc(32 * PGSIZE); // allocate 32 pages
    if (pgaccess(buf, 32, &abits) < 0) // cishi hai meiyou access, yinggai abits bubian , tongshi
    {
        err("pgaccess failed");
        buf[PGSIZE * 1] += 1;
        buf[PGSIZE * 2] += 1;
        buf[PGSIZE * 30] += 1; // write yishang 3 pages
        if (pgaccess(buf, 32, &abits) < 0)
        {
            err("pgaccess failed");
        }
        if (abits != ((1 << 1) | (1 << 2) | (1 << 30))) // dui ying weizhi ying bianhua
        {
            err("incorrect access bits set");
        }
        free(buf);
        printf("pgaccess_test: OK\n");
    }
}
```

2. 在 `kernel/riscv.h` 中增加 `PTE_A`, 并查阅资料发现他在页表信息的第六位



```
#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) //new
```

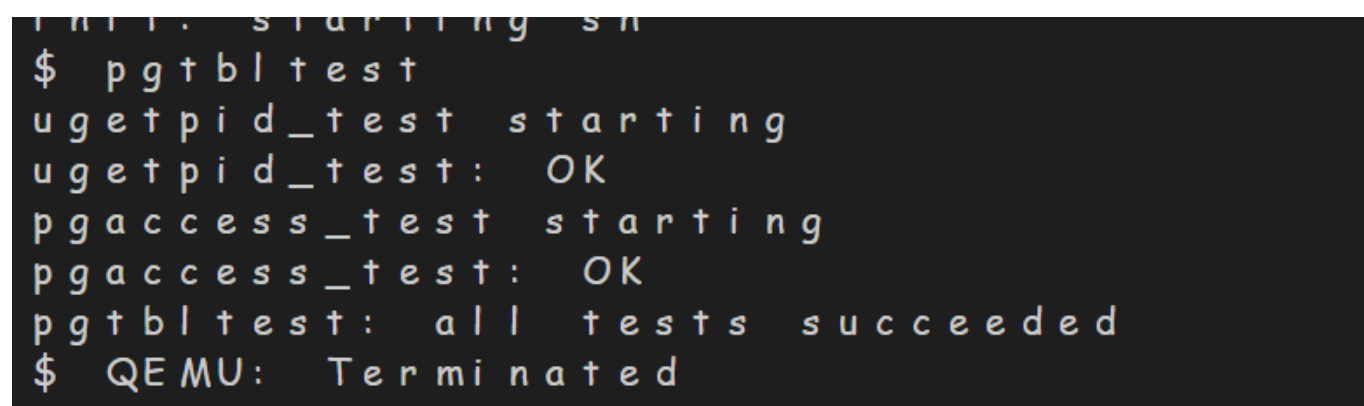
3. 补全 `sysproc.c` 中的 `sys_pgaccess()`, 先考虑参数传递, 由于 `pgaccess()` 传递了三个参数, 第一个和第三个是地址, 第二个是 `int`, 故分别用 `argaddr()`, `argint()`, `argaddr()`, 随后设置64位掩码用来存储记录哪一个页表被访问过了, 并通过 `myproc()` 获取当前进程。遍历每个页表, 通过 `walk()` 函数获取当前页表的信息, 如果被访问过, 就将掩码对应位置置1, 并清除 `PTE_A`, 将页表的地址指向下一个虚拟页的起始地址。最后通过 `copyout` 将 `maskbits` 穿回用户空间

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

    uint64 maskbits = 0;
    struct proc *p = myproc();
    pte_t *pte;
    for(int i=0; i<num; i++)
    {
        pte = walk(p->pagetable, addr, 0);
        if(pte==0)
        {
            panic("page not exists.");
            return -1;
        }
        if(PTE_FLAGS(*pte) & PTE_A)
```

```
    {  
        maskbits = maskbits|(1L<<i);  
        *pte &=(~PTE_A);  
    }  
    addr +=PGSIZE;  
}  
if(copyout(p->pagetable,abits,(char*)&maskbits,sizeof(maskbits))<0)  
{  
    return -1;  
}  
return 0;  
}
```

运行结果



```
init: starting sm  
$ pgtbltest  
ugetpid_test starting  
ugetpid_test: OK  
pgaccess_test starting  
pgaccess_test: OK  
pgtbltest: all tests succeeded  
$ QEMU: Terminated
```

实验中的问题与解决

在进行实验一时，一直出现usertrap()报错，经过gdb调试，发现当创建子进程后wait(&ret)会导致内陷,推测是初始化时出现问题

```
29         ugetpid_test()  
B+ 30     {  
31         int i;  
32  
33         printf("ugetpid_test sta  
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() != ugetpi  
45             err("mismatched PID  
46             exit(0);  
47     }
```

Thread 1.1 In: ugetpid_test L38 PC: 0x78

(gdb) p ret

\$1 = 6

(gdb) ☐

```
user/usys.S
7          ret
8          .global exit
9          exit:
10         li a7, SYS_exit
11         ecall
12         ret
13         .global wait
14         wait:
> 15        li a7, SYS_wait
16         ecall
17         ret
18         .global pipe
19         pipe:
20         li a7, SYS_pipe
21         ecall
22         ret
23         .global read
24         read:
25         li a7, SYS_read

In: wait          L15      PC: 0x4b2
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 1, ugetpid_test ()
    at user/pgtbltest.c:30
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) s
fork () at user/usys.S:5
(gdb) s
(gdb) s
(gdb) s
ugetpid_test () at user/pgtbltest.c:38
(gdb) s
(gdb) s
wait () at user/usys.S:15
(gdb) □
```

```

user/usys.S
[ No Source Available ]

In:                                L??      PC: 0x80006342
(gdb) n
(gdb) n
(gdb) n
(gdb) s
fork () at user/usys.S: 5
(gdb) s
(gdb) s
(gdb) s
ugetpid_test () at user/pgtbltest.c: 38
(gdb) s
(gdb) s
wait () at user/usys.S: 15
(gdb) s
(gdb) s
0x0000000080006342 in ?? ()
=> 0x0000000080006342: f5 9b andi a5,
a5, -3
(gdb) 

```

观察初始化分配页表的代码，发现在p->usyscallpage被分配空间之前，要先让p->pagetable获取页表，如果页表分配失败，即内存不足或其他错误，就会直接返回并且释放空间，这会导致p->usyscallpage没有被分配空

间，可能会导致后续访问缺页，从而导致trap

```
}

// An empty user page table.
p->pagetable = proc_pagetable(p);
if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
}

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

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

// Set up new context to start executing at forkret,
// which returns to user space.
memset(&p->context, 0, sizeof(p->context));
p->context.ra = (uint64)forkret;
```