

Lab - Page Tables 目录

1. Speed up system calls (easy)

- 1.1 实验目的
- 1.2 实验步骤
- 1.3 实验中遇到的问题和解决方法
- 1.4 实验心得

2. Print a page table (easy)

- 2.1 实验目的
- 2.2 实验步骤
- 2.3 实验中遇到的问题和解决方法
- 2.4 实验心得

3. Detect which pages have been accessed (hard)

- 3.1 实验目的
- 3.2 实验步骤
- 3.3 实验中遇到的问题和解决方法
- 3.4 实验心得

1. Speed up system calls (easy)

1.1 实验目的

- 优化系统调用 `getpid()`
- 了解内存空间的分配、映射与释放

1.2 实验步骤

1. 在 `user/ulib.c` 中已经定义了 `ugetpid()` 函数，使用 `USYSCALL` 映射来获取进程的 `PID`；当每个进程被创建时，在 `USYSCALL`（在 `kernel/memlayout.h` 中定义的虚拟地址）处映射一个只读页。在该页的开头，存储一个 `struct usyscall`（也在 `kernel/memlayout.h` 中定义），并将其初始化为存储当前进程的 `PID`。下面对这些代码进行分析：

- `user/ulib.c` 中的 `ugetpid()`：

定义了一个结构体 `usyscall` 的指针（定义见下），并通过 `USYSCALL` 映射来获取，返回所映射内容中存储的 `PID`。

```
// [user/ulib.c]
#ifdef LAB_PGTBL
int ugetpid(void)
{
    struct usyscall *u = (struct usyscall *)USYSCALL;
    return u->pid;
}
#endif
```

- `kernel/memlayout.h` 中的相关定义：

`TRAMPOLINE` 表示一个地址，表示从 `MAXVA - PGSIZE` 开始的一个 `PGSIZE` 大小的页空间，其中 `MAXVA` 表示虚拟内存空间栈顶。

`TRAPFRAME` 则是使用了 `TRAMPOLINE` 下方的一个页空间。

同理，`USYSCALL` 直接指向了一块页空间，在空间内存储着结构体 `usyscall` 的一个实例。

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

#ifdef LAB_PGTBL
#define USYSCALL (TRAPFRAME - PGSIZE)
struct usyscall {
    int pid; // Process ID
};
#endif
```

2. 首先在 `kernel/proc.h` 定义物理存储空间的指针，也即意味着要在进程创建时就初始化这样的一个指针，指向被分配的一块空间。但这里只是一个指针，说明还要实现以下功能：
 - 要分配空间，并让该指针指向该空间
 - 完成空间的映射

```
struct usyscall *usys; // 物理存储空间
```

3. 在 `kernel/proc.c` 的 `proc_pagetable()` 函数中进行映射操作：

```
pagetable_t proc_pagetable(struct proc *p)
{
    pagetable_t pagetable;

    // An empty page table.
    pagetable = uvmcreate();
    if (pagetable == 0)
        return 0;

    // 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;
    }

    // ADDED
    if (mappages(pagetable, USYSCALL, PGSIZE, (uint64)(p->usys), PTE_R |
PTE_U) < 0) {
        // 若映射失败，恢复上述页
        uvmunmap(pagetable, TRAMPOLINE, 1, 0);
        uvmunmap(pagetable, TRAPFRAME, 1, 0);
        uvmfree(pagetable, 0);
        return 0;
    }

    return pagetable;
}
```

同理，在页释放时也应一并解除这样的映射关联：

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

4. 在 `allocproc()` 中分配和初始化页面，要注意的是，不仅要进行分配，还要将 `PID` 移动至共享页中：

```

// Look in the process table for an UNUSED proc.
// If found, initialize state required to run in the kernel,
// and return with p->lock held.
// If there are no free procs, or a memory allocation fails, return 0.
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;
    }

    // ADDED
    if ((p->usys = (struct usyscall *)kalloc()) == 0) {
        freeproc(p);
        release(&p->lock);
        return 0;
    }
    memmove(p->usys, &p->pid, sizeof(int)); // 将pid移动至共享页中

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

    // 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;
    p->context.sp = p->kstack + PGSIZE;

    return p;
}

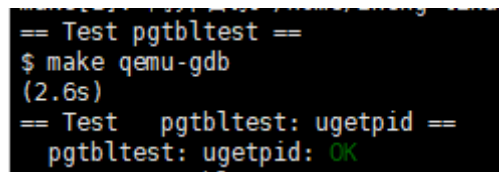
```

5. 在 freeproc() 中释放页面:

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

6. 运行 `grade` 程序进行测试，结果如下：



```
== Test pgtbltest ==
$ make qemu-gdb
(2.6s)
== Test pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
```

1.3 实验中遇到的问题和解决方法

- **问题：**测试时出现如下错误信息：

```
usertrap(): unexpected scause 0x000000000000000d pid=4
sepc=0x0000000000000490 stval=0x0000003fffffd000
```

解决方法：排查后发现是由于权限位设置错误导致，使页表允许用户访问即可。

- **问题：**不知道如何在 `proc_pagetable()` 中进行映射操作

解决方法：使用 `mappages()` 函数来在 `proc_pagetable()` 中进行页面映射。该函数用于在页表中插入映射，需要提供目标虚拟地址、物理地址和权限位。

1.4 实验心得

在本次实验中，我明白了可以通过在用户空间和内核之间共享只读的数据区域，来加快系统调用的执行速度、优化系统调用，并动手在 xv6 操作系统中实现这种优化，理解了如何在进程创建时进行页面映射、正确地设置权限位。此优化能够减少内核切换的开销，提高系统调用的执行效率。进一步思考，对于 `sysinfo()` 函数，也能够实现类似的优化，可以在进程创建时进行类似的页面映射以避免频繁的内核切换。实验过程中的探索和学习使我对操作系统的内核优化有了更深入的认识，也增强了我的解决问题的能力。

2. Print a page table (easy)

2.1 实验目的

- 编写 `vmprint()` 的函数，用于打印 RISC-V 页表内容
- 理解 xv6 的三级页表结构

2.2 实验步骤

1. 明确 xv6 使用三级页表结构，其中有三个级别的页表：

- 一级页表 (L1 Page Table)：也称为页全局目录 (Page Global Directory, PGD)，存放二级页表 (L2 Page Table) 的基址。
- 二级页表 (L2 Page Table)：也称为页目录 (Page Directory)，存放三级页表 (L3 Page Table) 的基址。
- 三级页表 (L3 Page Table)：存放实际的页表项，用于映射虚拟地址到物理地址。

每个页表是由 512 个页表条目 (Page Table Entries / PTE) 组成的数组，每个 PTE 包含一个 44 位的物理页码 (Physical Page Number/PPN) 和一些标志位。

其中有如下需要用到的标志位：

- PTE_V (Valid Bit)：表示该页表项是否有效，即是否有映射关系。
- PTE_R (Read Bit)：表示是否可读。
- PTE_W (Write Bit)：表示是否可写。
- PTE_X (Execute Bit)：表示是否可执行。

如果 PTE_V 为零，则表示该虚拟地址没有映射到物理地址，这时就不需要继续查找下一级页表，直接认为虚拟地址无效。

如果 PTE_V 为一，但 PTE_R、PTE_W 和 PTE_X 均为零，表示该虚拟地址映射到物理地址，但没有读、写或执行权限，这时认为是一级页表映射。

如果 PTE_V 为一，且 PTE_R 或 PTE_W 或 PTE_X 中至少有一个为一，表示该虚拟地址映射到物理地址，并且有相应的读、写或执行权限，这时继续查找下一级页表，即二级页表。

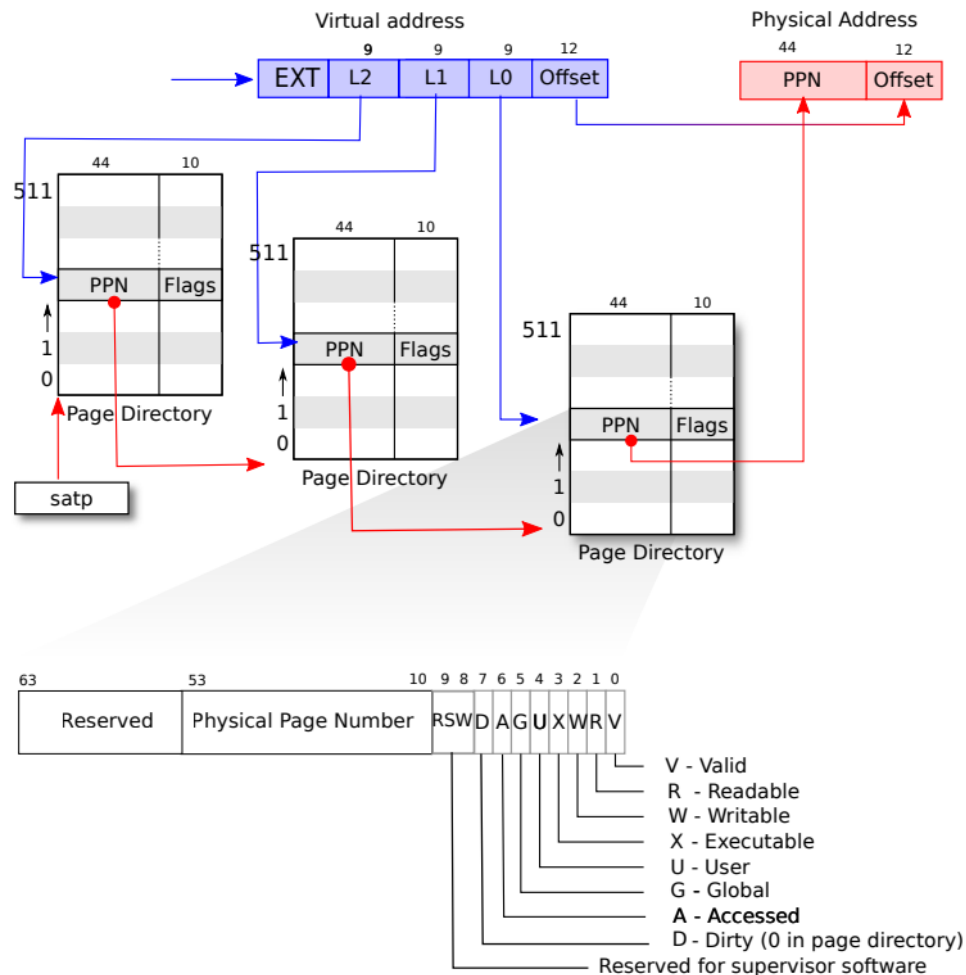


Figure 3.2: RISC-V address translation details.

2. 开始编写程序，在 `exec.c` 中加入 `vmprint()` 调用：

```
// ADDED
if(p->pid == 1){
    printf("page table %p\n", p->pagetable);
    vmprint(p->pagetable, 1);
}
```

3. 在 `kernel/defs.h` 中定义 `vmprint()` 的原型：

```
void vmprint(pagetable_t, int);
```

该函数接受两个参数，一是页表，二是当前页表层级。

4. 将 `vmprint()` 函数放置在 `kernel/vm.c` 中，参考 `freewalk()` 函数，使用递归的思想进入到每一级页表中并打印所需项：

```
void vmprint(pagetable_t pgtbl, int level)
{
    // there are 2^9 = 512 PTEs in a page table.
    for (int i = 0; i < 512; i++) {
        pte_t pte = pgtbl[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);
for (int j = 0; j < level; j++) {
    printf("..\n");
}
printf("%d: pte %p pa %p\n", i, pte, child);
vmprint((pagetable_t)child, level + 1);
}
else if (pte & PTE_V) {
    uint64 child = PTE2PA(pte);
    for (int j = 0; j < level; j++) {
        printf("..\n");
    }
    printf("%d: pte %p pa %p\n", i, pte, child);
}
}
}

```

5. 启动qemu模拟器便可看到自动调用了 `vmprint()` 函数，结果如下：

```

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

```

6. 运行 `grade` 测试程序，结果如下：

```

zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-page-tables$ ./grade-lab-pgtbl pte
make: "kernel/kernel" 已是最新。
== Test pte printout == pte printout: OK (1.1s)

```

2.3 实验中遇到的问题和解决方法

- **问题：**只输出一级和二级页表目录，第三级不打印（直接报错）

解决方法：没有判断处在第三级页表目录的PTE，其判断条件应该是 `(pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 1`。

2.4 实验心得

本次实验中，我成功实现了一个用于打印RISC-V页表的函数 `vmprint()`。该函数能够按照指定的格式输出页表的内容，包括PTE的索引、PTE位和物理地址。通过输出的信息，可以更好地理解页表的结构和层次关系。在不断的尝试和学习中，我理解了xv6系统页表的组织方式，包括顶层页表、中间页表和底层页表之间的关系。这使我对操作系统的内存管理有了更深入的理解。

3. Detect which pages have been accessed (hard)

3.1 实验目的

- 编写名为 `pgaccess()` 的系统调用，用于检测并报告用户访问的页

3.2 实验步骤

1. 在 `kernel/riscv.h` 中定义 `PTE_A`，即访问位，详情见上个实验的 Figure 3.2：

```
#define PTE_A (1L << 6) // bit of access
```

2. 鉴于包括添加到系统调用、用户内核接口等工作已经完成，只需在 `kernel/sysproc.c` 中实现 `sys_pgaccess()` 函数即可，要使用到 `kernel/vm.c` 中的 `walk()` 函数以访问正确的 PTE，代码如下：

```
#define PGACCESS_MAX_PAGE 32
int sys_pgaccess(void)
{
    // lab pgtbl: your code here.
    uint64 va, buf;
    int pgnum;

    // 解析参数
    argaddr(0, &va);
    argint(1, &pgnum);
    argaddr(2, &buf);

    if (pgnum > PGACCESS_MAX_PAGE)
        pgnum = PGACCESS_MAX_PAGE;

    struct proc *p = myproc();
    if (!p) {
        return -1;
    }

    pagetable_t pgtbl = p->pagetable;
    if (!pgtbl) {
        return -1;
    }

    uint64 mask = 0; // 位掩码
    for (int i = 0; i < pgnum; i++) {
        pte_t *pte = walk(pgtbl, va + i * PGSIZE, 0); // 访问PTE，检查
        if (*pte & PTE_A) {
            *pte &= (~PTE_A); // 复位
            mask |= (1 << i); // 标注第i个页是否被访问过
        }
    }

    // 复制到用户栈区
    copyout(p->pagetable, buf, (char *)&mask, sizeof(mask));

    return 0;
}
```

3. 运行 `grade` 测试程序，结果如下：

```
riscv64-linux-gnu-objdump -t kernel
== Test pgtbltest == (1.7s)
== Test pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
```

3.3 实验中遇到的问题和解决方法

- **问题：**出现的结果中，有些的访问块是上一次测试中的访问块

解决方法： `PTE_A` 只会置位而不会复位（在单个进程中）。在检查访问位之后，需要把该块的 `PTE_A` 复位。只有这样在下次调用 `pgaccess()` 时才能正确判断是否访问了该页。

- **问题：**忘记如何解析参数并检查用户访问的页

解决方法： 需要使用 `argaddr()` 和 `argint()` 来解析传递给 `pgaccess()` 系统调用的参数。然后，使用 `walk()` 函数找到当前进程中正确的PTE并检查访问位。

3.4 实验心得

本实验中，我成功实现了 `pgaccess()` 这一系统调用，该函数用于检测并输出用户访问的页。该系统调用接受三个参数：起始虚拟地址、需要检查的页数和输出位掩码，然后通过检查PTE的访问位来确定哪些页已被访问。在编程过程中，我温习了 `argaddr()`、`argint()`、`copyout()` 等函数的使用，同时也对于操作系统的内存管理和页面访问有了更深的理解