

Page Tables 实验报告



Tongji University

专 业：软 件 工 程

指 导 老 师：王 冬 青

年 级：大 学 二 年 级

学 号：2153061

姓 名：谢 嘉 麒

- 1. Speed up system calls (easy)
 - 1.1.实验目的
 - 1.2.实验步骤
 - 1.2.1.准备工作
 - 1.2.2.页表映射
 - 1.2.3.释放内存
 - 1.2.4.编译与检测
 - 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
 - 2.2.1.修改exec.c
 - 2.2.2.vmprint()函数实现
 - 2.2.3.添加函数声明
 - 2.2.4.编译与检测
 - 2.3.实验中遇到的问题和解决方法
 - 2.3.1.非法内存的访问
 - 2.3.2.递归终止条件和页表层级
 - 2.4.实验心得
- 3. Detecting which pages have been accessed (hard)
 - 3.1.实验目的
 - 3.2.实验步骤
 - 3.2.1.浏览测试程序
 - 3.2.2.加入sys_pgaccess系统调用
 - 3.2.3.实现函数access_check()
 - 3.2.4.编译与检测
 - 3.3.实验中遇到的问题和解决方法
 - 3.3.1.确定access_check()的返回值
 - 3.4.实验心得

1. Speed up system calls (easy)

1.1.实验目的

一些操作系统（如Linux）通过在一个用户空间和内核之间的只读域中共享数据来加速特定的系统调用。这一方法减少了在系统调用时的内核交错的需求。实验要求在每一个进程创建时，在USYSCALL上映射一个只读页面。在页面开始，存储一个struct usyscall，并且将其初始化存储当前进程的PID。本次实验已经在用户空间提供ugetpid()函数并且能够自动使用USYSCALL映射。需要完成映射页面的功能。

1.2.实验步骤

1.2.1.准备工作

在完成页面映射的功能之前，需要先阅读实验给出的函数和方法指导，同时对这些内容进行理解和本地适配。

由于实验需要存储struct usyscall，需要对相关内容进行修改：

在user/ulib.c中观察函数int ugetpid(void)。函数将进程地址USYSCALL强制类型转换为struct usyscall* 类型的指针变量，返回指向的结构体成员pid。因此可以推断需要在kernel/proc.h中的结构体struct proc中加入一个成员struct usyscall* ucall。

1.2.2.页表映射

在映射USYSCALL前，需要为成员ucall分配内存。因此在kernel/proc.c的函数allocproc()中，为ucall分配内存。之后才调用proc_pagetable()函数映射。

```
if (0 == (p->ucall = (struct usyscall*)kalloc()))
{
    freeproc(p);
    release(&p->lock);
    return 0;
}
p->pagetable = proc_pagetable(p);
...
p->ucall->pid = p->pid;
```

在这之后则修改kernel/proc.c的proc_pagetable()函数，提供页表映射内容。实验提到这一部分是只读权限，因此设置权限位 PTE_U | PTE_R，PTE只能在特权模式下使用。要注意并发读写的情况，因此最好加锁。

```
if (mappages(pagetable, USYSCALL, PGSIZE, (uint64)(p->ucall),
            PTE_U | PTE_R) < 0)
{
    uvmfree(pagetable, 0);
    return 0;
}
```

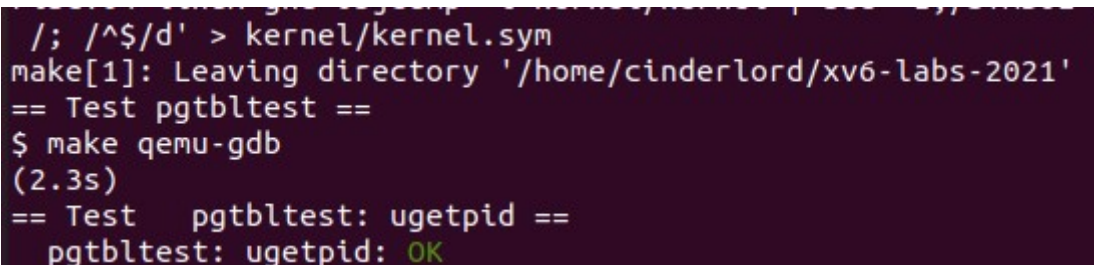
1.2.3.释放内存

完成以上步骤后，需要考虑到在进程结束后释放内存。浏览实验源文件得知释放进程内存的函数为freeproc()，则在其中添加释放页面的代码即可。

具体释放方法为当p->usyscall存在，则调用kfree()函数进行释放，同时将p->usyscall置0。

1.2.4.编译与检测

在完成以上步骤后，进入xv6目录并启动终端，执行指令make qemu，运行检测指令，看到预期输出如下：



```
;/^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/cinderlord/xv6-labs-2021'
== Test pgtbltest ==
$ make qemu-gdb
(2.3s)
== Test   pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
```

1.3.实验中遇到的问题和解决方法

本次实验主要难点在于理解系统调用的方法和思想，按照其思想进行具体的代码实现即可。

1.4.实验心得

通过本次实验，笔者学习并理解了一些操作系统进行加速系统调用的具体方法，并且对这种方法的具体实现，包括数据结构，函数依赖和运行等方面，都有了更深的印象。

2. Print a page table (easy)

2.1.实验目的

为了可视化RISC-V页表，同时也帮助未来的debug，实验要求书写一个能够打印页表内容的函数。具体要求为定义一个叫做vmprint()的函数，这个函数接收一个pagetable_t类型的参数，并且以实验要求的形式打印页表。

2.2.实验步骤

2.2.1.修改exec.c

vmprint()运行需要在kernel/exec.c的exec()函数中添加vmprint()函数的调用。观察这个函数得知，函数获取两个参数，一个是页表，另一个是层数。在exec函数中添加if语句，当p->pid等于1时，利用printf打印p->pagetable并且调用vmprint()。

2.2.2.vmprint()函数实现

需要在kernel/vm.c中实现函数vmprint()，具体实现思路如下：

- (1) 函数接受两个参数pagetable_t类型的pagetable和int类型的level;
- (2) 首先静态定义char* 类型的三种level，即当前递归的深度来标识页表的层级。
- (3) level1, level2, level3分别为"..", "...", "...";
- (4) 定义char* 类型变量buf作为指向缩进字符的指针。
- (5) 接下来则进入主体for循环，遍历页表的所有512项：
 - a. 获取当前遍历位置的页表的第i项，存储在变量pte中;
 - b. 当前页表有效位PTE_V为1且递归深度level不大于3时，页表项有效，继续递归打印子页表。
 - c. 获取当前页表指向的子页表的物理地址，存放在child变量中，并且利用printf函数打印当前页表项信息，包括缩进字符，索引i，页表项的值pte，子页表的物理地址child。
 - d. 继续递归调用vmprint函数递归打印子页表信息。

2.2.3.添加函数声明

需要在defs.h中添加vmprint的函数声明。

2.2.4.编译与检测

完成以上步骤后，进入xv6目录并在终端中执行指令make qemu进行编译，编译完成后即可看到预期输出，如下：

```
xv6 kernel is booting

hart 1 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
hart 2 starting
init: starting sh
```

2.3.实验中遇到的问题和解决方法

本次实验在确定实验目的和页表信息后容易理解，但是在代码实现时仍需要警惕。

2.3.1.非法内存的访问

vmprint()函数递归遍历页表时，当页表指向的物理地址是非法的或者未映射的内存区域，则会导致访问非法内存而系统崩溃。因此需要添加页表有效的判断。

2.3.2.递归终止条件和页表层级

设置页表的层级深度也是递归函数支持的最大深度3，放置发生无线递归的现象，同时也避免页表层级错误。

2.4.实验心得

本次实验使笔者进一步了解了页表项具体的信息内容和访问方法，也在代码纠错时更加注意代码的健壮性和安全性问题，对于谨慎处理页表的访问有了进一步的认识。

3. Detecting which pages have been accessed (hard)

3.1.实验目的

很多现代编程语言具备内存垃圾回收的功能，而这一功能的实现需要得到一个页面自从上一次检查后是否被访问这一信息。利用纯软件实现这一功能效率低下，而利用页表的硬件机制和操作系统结合的方法则相对高效。实验要求添加一个新的xv6功能来检测并报告这一信息。

需要实现pgaccess()系统调用，这一系统调用接收三个参数：第一个参数是需要检查页面的初始地址；第二个参数是从初始地址向后检查的页面数量；第三个参数是指向存储结果的位向量的首地址指针；

3.2.实验步骤

3.2.1.浏览测试程序

实验给出了测试程序user/pgbtltest.c，可以看到给buf分配了32页，因此用一个32位的int型变量就能来存储哪一页已经被访问过的信息。

3.2.2.加入sys_pgaccess系统调用

首先需要根据3.2.1.的推断在riscv.h中加入宏定义PTE_A ($1L \ll 6$);

之后再kernel/sysproc.c中加入系统调用sys_pgaccess:

- (1) 定义测试程序中的buf地址为start_addr，32位int型变量amount和abits的地址buffer;
- (2) 分别利用argaddr(), argint(), 和argaddr()三个函数调用来获取三个参数，如果三个参数返回值有一个(或一个以上)小于零，则返回-1;
- (3) 定义proc* 类型指针p，用myproc()函数获取当前进程。
- (4) 调用access_check()函数(接下来会实现)来获取掩码并存入mask变量中，用于检查地址对应的物理页面是否已经被访问。

3.2.3.实现函数access_check()

这一函数用于检查指定的虚拟地址参数对应的物理页面是否已经被访问，并将访问情况以位掩码的形式来返回。

在kernel/vm.c中实现函数access_check(), 具体思路如下:

- (1) 首先检查虚拟地址va是否超过了最大虚拟地址MAXVA，超过则发生内核恐慌;
- (2) 定义变量mask来存储位掩码;
- (3) 循环检查va所在的每一页(32)。每次循环更新va的值为下一页的虚拟地址;
- (4) 每次循环开始时，定义指向页表项的指针pte并初始化0;
- (5) 定义pagetable_t类型的变量p并初始化为pagetable，用于在页表中进行查找;
- (6) 嵌套循环，从最高级页表项依次查找va对应的页表项，并将指针pte指向该页表项。若页表项有效(PTE_V标志位为1)，则将p更新为下一级页表的地址;
- (7) 查找完成后，检测最终找到的页表项是否有效(PTE_V为1)且已经被访问(PTE_A为1);
- (8) 满足条件则将对位掩码置1，表示页面已经被访问;
- (9) 将页表项的访问标志PTE_A清楚，以便下次访问时能够重新检测是否已经被访问过;
- (10) 返回位掩码，每个位表示一个页面，为1则表示页面已经被访问，为0则表示页面没有被访问;

3.2.4.编译与检测

在xv6目录的终端下用make qemu进行编译，执行指令make grade检测。得到预期结果如下:

```
make[1]: Leaving directory '/home/cinderlord/xv6-labs/xv6-labs-2021-lab3'
== Test pgtbltest ==
$ make qemu-gdb
(1.9s)
== Test pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.7s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(245.1s)
== Test usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
cinderlord@ubuntu:~/xv6-labs/xv6-labs-2021-lab3$
```

3.3.实验中遇到的问题解决方法

3.3.1.确定access_check()的返回值

该用哪种类型的变量来确定地址对应的物理页面是否被访问，是实验要解决的一个问题。因此需要在源代码中寻找线索。通过观察测试程序user/pgtbltest.c文件，可以看到buf有32位，因此选择使用int型变量来存放这一结果。

3.4.实验心得

通过本次实验，笔者对RISC-V的页表机制有了更深的了解。实验大致介绍了RISC-V提供的分页机制和操作系统配合的过程，包括页面映射，访问页检测等。同时具体的代码实现也揭露了关于RISC-V页表的很多细节，如具体的映射方法，页表的有效位等。