Lab 3 Page Tables

目录

[1. Speed up system calls (easy) 1](#_Toc174491018)

[1.1 实验目的 1](#_Toc174491019)

[1.2 实验步骤 1](#_Toc174491020)

[1.3 实验结果 5](#_Toc174491021)

[1.4 遇到的问题 5](#_Toc174491022)

[1.5 实验心得 6](#_Toc174491023)

[2. Print a page table (easy) 7](#_Toc174491024)

[2.1 实验目的 7](#_Toc174491025)

[2.2 实验步骤 7](#_Toc174491026)

[2.3 实验结果 10](#_Toc174491027)

[2.4 实验中遇到的问题和解决方法 11](#_Toc174491028)

[2.5 实验心得 12](#_Toc174491029)

[3. Detect which pages have been accessed (hard) 12](#_Toc174491030)

[3.1 实验目的 12](#_Toc174491031)

[3.2 实验步骤 13](#_Toc174491032)

[3.3 实验结果 14](#_Toc174491033)

[3.4 遇到的问题 14](#_Toc174491034)

[3.5 实验心得 15](#_Toc174491035)

[4. 测试结果 16](#_Toc174491036)

Speed up system calls (easy)

* 1. 实验目的
* 实现在xv6操作系统中优化特定系统调用（如getpid( )）的功能。
* 通过在用户空间和内核之间共享只读页面，消除执行系统调用时的内核切换，从而提高系统调用的执行速度。
* 映射一个只读页面，并在页面开头存储相关的信息，例如当前进程的PID。
* 为用户空间提供适当的接口（如ugetpid( )），以便自动使用共享页面进行系统调用。
* 验证优化的有效性，通过运行pgtbltest测试用例，确保优化后的系统调用能够成功执行。
  1. 实验步骤
* 原有的 user/ulib.c 中已经定义好了 ugetpid( ) 函数,通过 USYSCALL 映射来获取进程的 PID :

#ifdef LAB\_PGTBL

int

ugetpid(void)

{

struct usyscall\* u = (struct usyscall\*)USYSCALL;

return u->pid;

}

#endif

* kernel/memlayout.h 中定义的虚拟地址 USYSCALL ，映射了一个只读页，存储了一个 struct usyscall ,并将器初始化为存储当前进程的 PID

#define TRAMPOLINE (MAXVA - PGSIZE)

#define TRAPFRAME (TRAMPOLINE - PGSIZE)

#ifdef LAB\_PGTBL

#define USYSCALL (TRAPFRAME - PGSIZE)

struct usyscall {

int pid; // Process ID

};

#endif

TRAMPOLINE ，指向内核的跳板代码页面。跳板代码用于在从用户态切换到内核态时保存和恢复 CPU 寄存器。 MAXVA 表示最大的虚拟地址， PGSIZE 表示页面大小。因此， TRAMPOLINE 位于虚拟地址空间的最高页面。

TRAPFRAME ，指向保存当前进程陷阱帧（trapframe）的页面。陷阱帧包含处理器的寄存器状态，当发生陷阱（如系统调用或中断）时，处理器的状态会被保存到陷阱帧中。 TRAPFRAME 位于TRAMPOLINE 的下一个页面。

USYSCALL ，指向用户态系统调用共享页面。在这个页面中存储了 struct usyscall 结构，用于在用户空间和内核空间之间共享系统调用相关的信息，如当前进程的PID。 USYSCALL 位于TRAPFRAME 的下一个页面。

struct usyscall 结构体用于存储系统调用相关的信息，即当前进程的PID。

* 在kernel/proc.c/proc\_pagetable()中添加映射：

pagetable\_t

proc\_pagetable(struct proc\* p)

{

pagetable\_t pagetable;

// An empty page table. 创建空页表

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. 映射跳板代码页

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. 映射陷阱帧页

if (mappages(pagetable, TRAPFRAME, PGSIZE,

(uint64)(p->trapframe), PTE\_R | PTE\_W) < 0) {

uvmunmap(pagetable, TRAMPOLINE, 1, 0);

uvmfree(pagetable, 0);

return 0;

}

// 添加映射 映射用户系统调用页

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;

}

// 同时释放时也要解除这样的映射

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);

}

* 需要在 kernel/proc.h 的 struct proc 中定义物理存储空间的指针：

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 context context; // swtch() here to run process

struct file\* ofile[NOFILE]; // Open files

struct inode\* cwd; // Current directory

char name[16]; // Process name (debugging)

// 添加物理空间指针

struct usyscall\* usys; // 物理存储空间

};

* 在 allocproc( ) 函数中分配一个新页面，并将其映射到 USYSCALL 地址,还要将 PID 移动至共享页中

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;

}

// 将PID移动到共享页中 begin

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

// end

// 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));

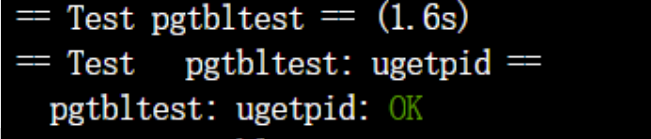
p->context.ra = (uint64)forkret;

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

return p;

}

* 1. 实验结果



* 1. 遇到的问题
* 问题：

具体实现过程不清楚

* 解决方法：

我们已经有了一个allocproc函数，作用是分配一个物理页面来存储一些可以直接从用户态读取的信息。这个函数会调用kalloc来获取一个物理页面的指针。尽管这个指针是虚拟地址，但由于内核页表的直接映射机制，我们可以将它当作物理地址来使用。

然后，我们将需要加速访问的数据（pid）直接放入这个页面，并在进程信息结构体中保持一个指向该页面的指针。需要注意的是，在这个时候，这个页面只是存在而已，并没有被映射到进程的地址空间中，因此在页表中是无法访问这个页面的。allocproc会调用proc\_pagetable函数，这个函数负责创建进程的页表，并将特殊的几个页面（trampoline、trapframe和我们的加速页面）映射到页表中。

这几个特殊页面必须已经被分配好，也就是说它们已经存在于物理内存中，我们在这里只是建立了映射关系。trampoline是内核代码的一部分，始终存在于内存中，而trapframe和我们的加速页面在之前的allocproc中已经被分配好了，所以它们也是存在的。因此，在proc\_pagetable中，我们使用mappages函数逐一将它们映射到进程的页表中。

完成这些后，一个进程在用户态下可以直接访问所谓的USYSCALL虚拟地址。当这个地址经过多级页表的翻译时，它会索引到我们分配的加速页面，从而直接获取到所需的数据。

* 1. 实验心得

在本次实验中，我通过实现用户态获取进程ID（PID）的功能，深入理解了操作系统中用户态与内核态的交互。通过 USYSCALL 地址实现了用户态与内核态共享进程的PID。这让我学会了如何利用页表实现内存保护和数据共享。

在 proc\_pagetable 函数中，我添加了 USYSCALL 的映射，深入理解了页表在进程内存管理中的作用。在 allocproc 函数中，为每个新进程分配了共享页，用于存储PID。这让我更好地理解了操作系统中的动态内存管理和进程初始化。

Print a page table (easy)

* 1. 实验目的
* 理解页表在操作系统中的作用以及其结构和组织方式。
* 学习如何遍历和打印页表的内容，以便可视化和调试。
* 实现一个名为vmprint()的函数，用于打印给定页表的内容。
* 理解页表的层次结构和索引，以及如何从页表中提取PTE的信息。
* 确定有效的PTE并根据其内容打印页表的格式化输出。
  1. 实验步骤
* 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。

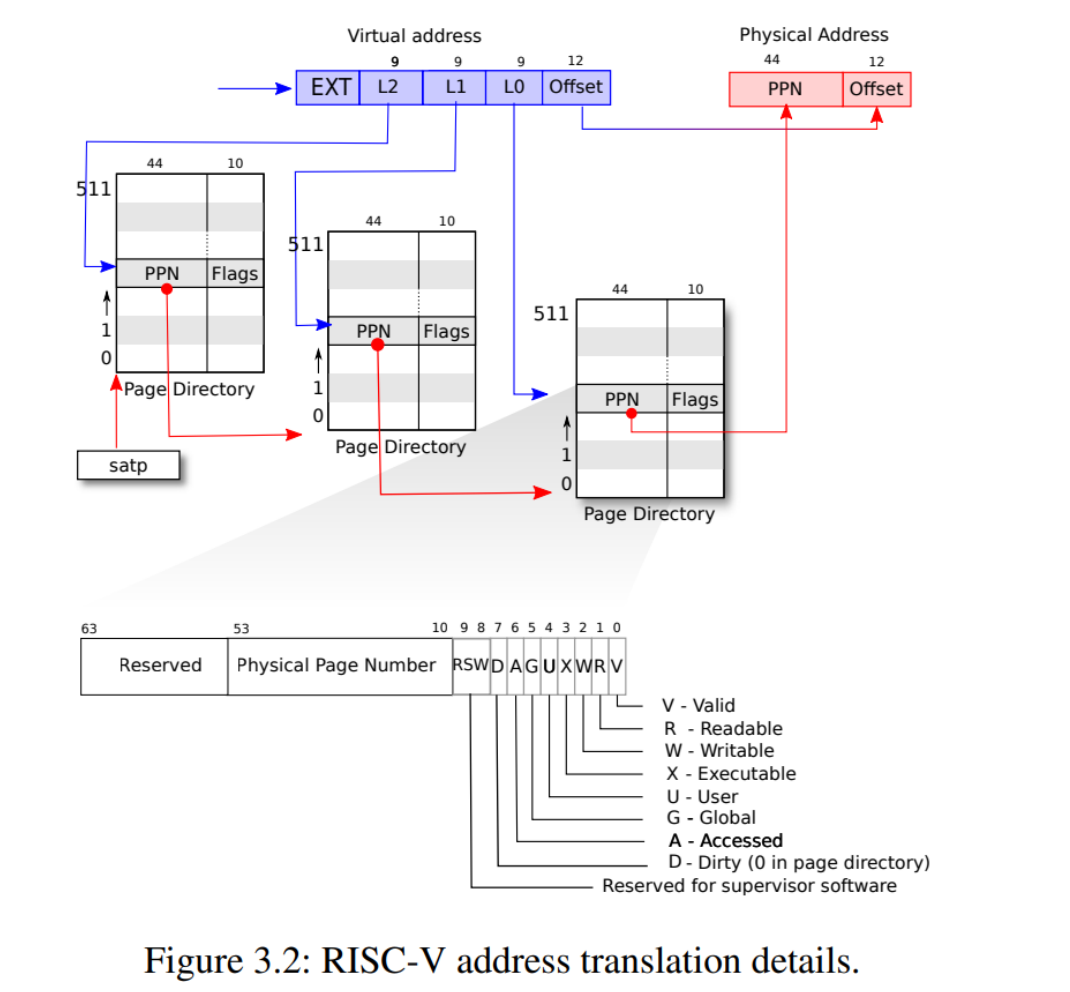
PTE\_W (Write Bit)：表示是否可写。

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。



* 在 kernel/vm.c 中定义 vmprint() 函数，该函数接收一个 pagetable\_t 类型的参数，并以指定格式打印页表的内容。

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("..");

}

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("..");

}

printf("%d: pte %p pa %p\n", i, pte, child);

}

}

}

* 修改 kernel/defs.h ，声明 vmprint 函数原型：确保在其他文件中可以调用该函数。

void vmprint(pagetable\_t, int);

* 在 exec.c 中调用 vmprint() ：在 if(p->pid==1) 条件下，调用 vmprint(p->pagetable) 打印第一个进程的页表。

// Commit to the user image.

oldpagetable = p->pagetable;

p->pagetable = pagetable;

p->sz = sz;

p->trapframe->epc = elf.entry; // initial program counter = main

p->trapframe->sp = sp; // initial stack pointer

proc\_freepagetable(oldpagetable, oldsz);

// 添加vmprint()调用

if (p->pid == 1) {

printf("page table %p\n", p->pagetable);

vmprint(p->pagetable, 1);

}

return argc; // this ends up in a0, the first argument to main(argc, argv)

bad:

if (pagetable)

proc\_freepagetable(pagetable, sz);

if (ip) {

iunlockput(ip);

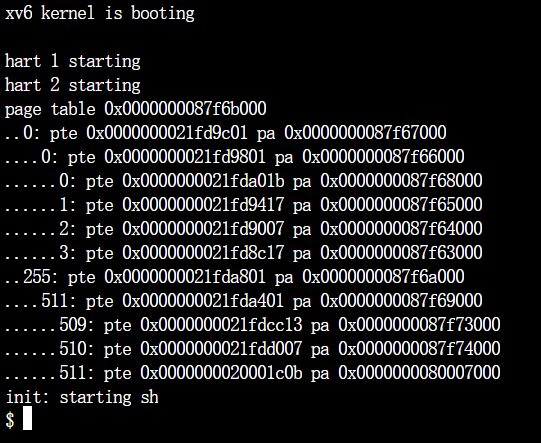
end\_op();

}

return -1; }

* 1. 实验结果





* 1. 实验中遇到的问题和解决方法
* 问题：

freewalk源码理解上有点困难

* 解决方法：

耐心看代码，仔细阅读之后我发现，函数首先遍历页表中的每个页表条目（PTE）。对于每个PTE，它检查是否设置了有效位（ PTE\_V ）和读/写/执行位（ PTE\_R 、 PTE\_W 、 PTE\_X ）。如果有效位被设置且读/写/执行位未被设置，则该PTE指向一个下级页表。在这种情况下，函数会递归调用自身来释放下级页表所占用的内存，并将当前PTE清零。如果有效位被设置且读/写/执行位中的任意一个被设置，则会调用 panic 函数报告错误。在遍历完所有PTE后，函数使用 kfree 函数释放页表所占用的内存。当我真正看懂了freewalk函数利用页表项实现递归访问其他页表的原理之后，我就明白了这个实验的意义，同时，因为因为xv6采用的是三级页表，所以只需递归三层就可以实现实验所要求的功能。

* 1. 实验心得

在这次实验中，我深入理解了xv6操作系统中页表的结构及其实现，并通过实践学习了页表遍历和打印的技术。我明确了xv6使用的三级页表结构，包括L1、L2、L3页表的组织方式和作用。每级页表通过页表项（PTE）指向下一级或直接映射到物理地址，这让我更好地理解了虚拟地址到物理地址的映射过程。

在 kernel/vm.c 中实现了 vmprint 函数，用于递归打印页表结构。这一过程让我掌握了如何通过遍历页表来展示内存映射关系，并且加深了对PTE标志位（如PTE\_V, PTE\_R, PTE\_W, PTE\_X）的理解。将 vmprint 函数集成到 exec.c 中，并在特定条件下调用，打印第一个进程的页表内容。通过这一操作，我学会了如何将调试工具嵌入操作系统代码中，从而更好地观察和理解操作系统的行为。

Detect which pages have been accessed (hard)

* 1. 实验目的
* 理解操作系统的页面访问机制
* 系统调用的实现和参数解析
* 内核和用户空间的数据交换
  1. 实验步骤
* 阅读 user/pgtlbtest.c 中的 pgaccess\_test() 函数，了解 pgaccess 的使用方法。
* 在 kernel/riscv.h 中定义PTE\_A（访问位）#define PTE\_A (1L << 6) // bit of access
* 在 kernel/sysproc.c 中实现 sys\_pgaccess() 函数。

#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

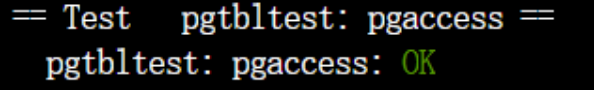
copyout(p->pagetable, buf, (char\*)&mask, sizeof(mask));

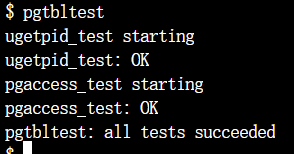
return 0;

}

sys\_pgaccess 用于检测和报告指定范围内内存页访问情况的系统调用实现。接收三个参数：起始虚拟地址、页数以及存储结果的用户地址。函数通过检查页表项的访问位，记录哪些页被访问过，并将结果以位掩码的形式存储在用户提供的缓冲区中。同时，函数会重置访问位，以便下次调用时能够检测新的访问情况。

* 1. 实验结果





* 1. 遇到的问题
* 问题：

不清楚哪些页表被访问了

* 解决方法：

每个 PTE 有个 PTE\_A 位，该位被置 1 则说明被访问过，该位被置 0 则说明没被访问过。置位操作有硬件完成，无需我们考虑。但是，硬件只能做到置位，无法做到复位。因此每次 sys\_pgacess 时要手动将 PTE\_A 复位 0。

* 1. 实验心得

我深入理解并实践了xv6操作系统中页表访问位的管理与检测，通过实现 sys\_pgaccess 函数， 通过阅读 pgaccess\_test 函数并实现 sys\_pgaccess，我深入理解了页表项中的访问位（PTE\_A）。访问位用于标识内存页是否被访问过，这在操作系统中非常重要，尤其是在实现页面替换算法时。

在 kernel/sysproc.c 中实现 sys\_pgaccess 函数，并通过解析参数、遍历页表和检测访问位，我掌握了如何在xv6中实现系统调用。这让我理解了操作系统如何在用户态与内核态之间传递信息，并通过页表管理进行内存访问控制。通过 copyout 函数将位掩码结果传递回用户空间，我进一步理解了如何安全地在内核与用户空间之间传递数据。

测试结果

