xv6 Lab3

刘鸣霄 自 12 2021010584

6 月 2 日

xv6 lab3-page tables 帮助我们熟悉操作系统中一个重要的概念：页表。

1. Speed up system call

这个题通过给每个进程额外提供一个特殊的页：usyscall page，其虚拟地址是 USYSCALL，来给某些系统调用加速。这页是用户和内核的一个共享只读页，存储一个结构体 struct usyscall，包含与该进程相关的信息，在这个题中，是进程的 pid。在进程需要 pid 时可以通过 USYSCALL 直接读取，而避免调用 getpid 向内核切换，从而给系统调用加速。

这个功能通过以下几个步骤来实现，修改 kernel/proc.h 和 kernel/proc.c：

1) 在 struct proc 中添加数据成员 usyscall，其是进程的 usyscall page 的指针

```
struct usyscall *usyscall;   // 新增：进程的 usyscall page 指针
```

2) 在 allocproc 函数中，完成该页的分配和初始化

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

// 新增：初始化 usyscall->pid
p->usyscall->pid = p->pid;
```

3) 在创建页表时建立虚拟地址 USYSCALL 到 usyscall 页的映射，这个页应该是用户可以访问，并且是只读的。调用 mappages 函数实现这一映射。在 proc_pagetable 函数中添加：

```
//新增:map the usyscall page just below the trapframe page
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;
}
```

4) 处理页的释放。在 freeproc 函数中添加：

```
// 新增：释放进程的 usyscall page
if(p->usyscall)
  kfree((void*)p->usyscall);
p->usyscall = 0;
```

在 proc_freepagetable 中添加：

```
uvmunmap(pagetable, USYSCALL, 1, 0); // 新增：清除 USYSCALL 页的 PTE
```

该部分完成。

问题：还有哪个 xv6 系统调用可以通过这种共享页的方式进行加速？如何实现？

我能想到的一个系统调用是 fstat，即获取打开的文件的信息。在调用 open 打开文件时，将文件的信息存放在 usyscall page 中。之后获取文件信息就可以直接从 USYSCALL 地址处读取。

2. Print a page table

在内核中实现一个函数 vmprint，可以打印页表。vmprint 的参数是待打印页表的物理地址。

根据 xv6 的手册，xv6 采用三级页表。每一级页表占一页 4KB，一个 PTE 为 64 位，共有 2^9 个 PTE。每一个虚拟地址的 38-12 位每 9 位表示对应级页表 PTE 的索引，而后 12 位是页内偏移量。每个 PTE 中 53-10 位对应的是下一级页表或者对应物理页的物理页号，而后 10 位是一些标记位，宏 PTE2PA 实现了 PTE 到物理地址的转换。

实现：

使用递归在三级页表中不断深入。在一张页表中遍历所有 PTE，找到 PTE_V 位不为 0 的 PTE 进行打印，如果它不是最后一级页表，递归深入到下一级页表进行遍历。

在 kernel/vm.c 函数中添加以下代码。vmprint 是入口函数，其参数 pagetable 是第一级页表的物理地址。_vmprint 是递归函数，参数 level 表示页表的级数。

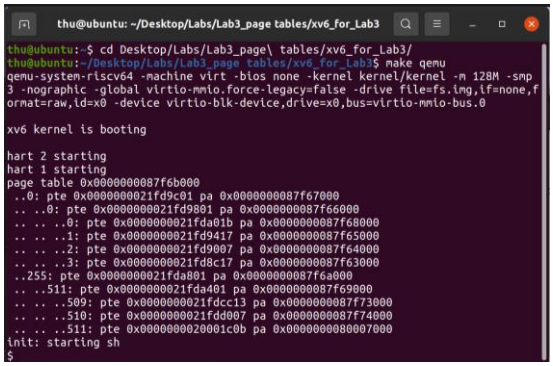
```
// 打印页表的递归函数
void
_vmprint(pagetable_t pagetable, int level){
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V){ // 如果页表项有效则打印
      uint64 pa = PTE2PA(pte); // 获得物理地址
      for(int i = 0; i < level; i++){
        printf(" .."); // 打印级数
      }
```

```
      printf("%d: pte %p pa %p\n", i, pte, pa); // 打印页表项
      if(level < 3){ // 如果不是最后一级（第三级）则递归打印下一级
        _vmprint((pagetable_t)pa, level + 1);
      }
    }
  }
}

// 新增：打印页表
void
vmprint(pagetable_t pagetable){
  printf("page table %p\n", pagetable); // 打印参数
  _vmprint(pagetable, 1); // 调用递归函数打印页表
}
```

在 exec 中添加相应的代码，发现 xv6 启动时会打印 1 号进程 init 的页表。该部分完成。

问题：解释打印的结果。



参考 xv6 手册，尤其是 exec 代码和图 3-4。

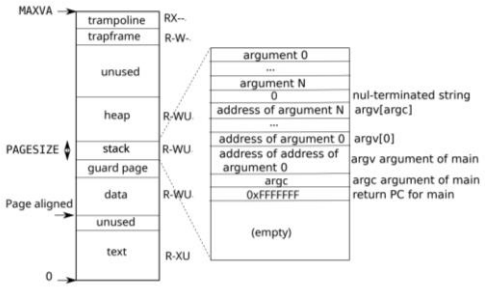


Figure 3.4: A process's user address space, with its initial stack.

0-0 页表：

0,1 页：从 init 的 ELF 文件加载的 segments。包含 init 的 text 和 data。

3 页：init 的用户栈。

2 页：用户栈的保护页。从 PTE 中观察到其写和读位都为 0，因此这页不能读写。

255-511 页表：

509,510,511 页：分别是 usyscall, trapframe, trampoline。

3. Detect which pages have been accessed

实现一个系统调用 pgaccess，收集那些 PTE_A 标志位为 1 的页，将结果存储在一个位图(bitmask)中。pgaccess 的参数是：起始虚拟地址，页面数和一个用于输出的整数。

查看程序 pgaccess_test。

```
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");
}
```

这个程序申请了 32 个页的内存，接着访问第 1,2,30 页，之后调用 pgaccess 检查这 32 个页。由于只有第 1,2,30 页被访问，pgaccess 返回的掩码应该是 1,2,30 位为 1，其余位为 0。之后这个程序检查是不是这个结果。

下面实现系统调用 pgaccess。它的系统操作号和槽等一些工作已经事先添加好了。

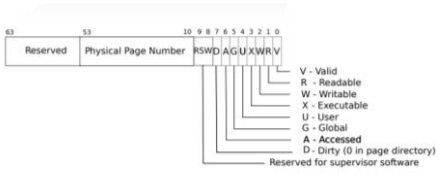


Figure 3.2: RISC-V address translation details.

根据 xv6 手册和 RISCV 手册，PTE_A 规定是第六位，因此在 kernel/riscv.h 中添加：

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

下面在 sysproc.c 中实现 sys_pgaccess:

```
int
```

```
sys_pgaccess(void)
{
  uint64 va, result;
  int sz;
  argaddr(0, &va);
  argint(1, &sz);
  argaddr(2, &result); // 从用户空间获得参数

  int buffer = 0; // 缓冲区
  struct proc* p = myproc(); // 陷入前的进程

  if(sz > 32){ // 最多获取 32 个页面
    return -1;
  }
  for(int i = 0; i < sz; ++i){
    // 获得各个虚拟地址对应的 PTE
    pte_t* pte = walk(p->pagetable, va + i * PGSIZE, 0);
    if(pte && (*pte & PTE_A)){ // 如果 PTE 存在并且 PTE_A 位为 1
      buffer |= 1 << i; // 设置缓冲位图
      *pte &= ~PTE_A; // 之后 PTE 的 PTE_A 位置 0
    }
  }

  // 将结果拷贝到用户空间
  copyout(p->pagetable, result, (char*)&buffer, sizeof(buffer));
  return 0;
}
```

首先使用 argaddr 和 argint 获取三个参数，定义整数 buffer 用于临时存储位图，指针 p 指向陷入前的进程。

之后检查页面数。由于输出参数为 int 为 32 位，故最多能检查 32 个页面，超出则返回-1 表示错误。

之后遍历每个虚拟地址。使用 walk 获得虚拟地址对应的（最后一级页表的）PTE。如果 PTE 存在且 PTE_A 位为 1，则在 buffer 中设置对应位，并将该 PTE 的 PTE_A 位置 0。

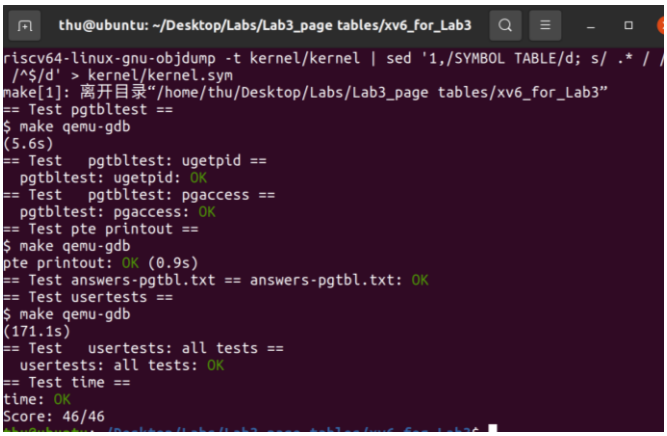
最后调用 copyout 将结果拷贝到用户空间。

该部分完成。

4. Time

该实验用时 4 小时，包括读手册的时间。这个实验比较好做，但是却详细地展示了操作系统中页表的概念与作用。

5. Make grade


thu@ubuntu: ~/Desktop/Labs/Lab3_page tables/xv6_for_Lab3
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /
 /^$/d' > kernel/kernel.sym
make[1]: 离开目录“/home/thu/Desktop/Labs/Lab3_page tables/xv6_for_Lab3”
== Test pgtbltest ==
$ make qemu-gdb
(5.6s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(171.1s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46