OS Lab 4

page tables

王世炟

PB20151796

November 25, 2022

Part 1: 实验要求

在本实验中，我们将探索页表并对其进行修改，以加快某些系统调用并检测哪些页面已经被访问。

Part 2: Print a page table

2.1 问题简述

本部分要求我们可视化 RISC-V 页表，写一个函数输出页表的内容。

2.2 知识基础

一个 RISC-V 页表在逻辑上是一个 $2^{27}(134,217,728)$ **页表项**（Page Table Entry, PTE）的数组。每个 PTE 包含一个 44 位的**物理页号**（Physical Page Number，PPN）和一些**标志位**。分页硬件通过利用 39 位中的高 27 位索引到页表中找到一个 PTE 来转换一个虚拟地址，并计算出一个 56 位的物理地址，它的前 44 位来自于 PTE 中的 PPN，而它的后 12 位则是从原来的虚拟地址复制过来的，在逻辑上可以把页表看成是一个简单的 PTE 数组。

如 **图** 1 所示，实际转换分三步进行。一个页表以三**层树**的形式存储在物理内存中。树的根部是一个 4096 字节的页表页，它包含 512 个 PTE，这些 PTE 包含树的下一级页表页的物理地址。每一页都包含 512 个 PTE，用于指向下一个页表或物理地址。分页硬件用 27 位中的顶 9 位选择根页表页中的 PTE，用中间 9 位选择树中下一级页表页中的 PTE，用底 9 位选择最后的 PTE。

每个 PTE 包含**标志位**，告诉分页硬件如何允许使用相关的虚拟地址。***PTE_V*** 表示 ***PTE*** 是否存在：如果没有设置，对该页的引用会引起异常（即不允许）。***PTE_R*** 控制是否允许指令读取到页。***PTE_W*** 控制是否允许指令向写该页。***PTE_X*** 控制 CPU 是否可以将页面的内容解释为指令并执行。***PTE_U*** 控制是否允许用户模式下的指令访问页面；如果不设置 ***PTE_U***，PTE 只能在监督者模式下使用。

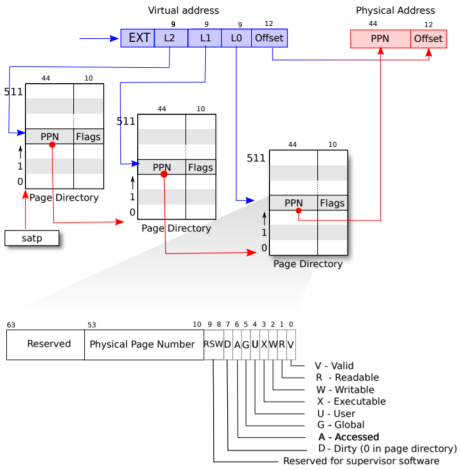

Figure 3.2: RISC-V address translation details.

图 1: RISC-V address translation details.

2.3 实验步骤

(1) **分析** freewalk() **函数**

根据所给提示，首先分析 kernel/vm.c 中的 freewalk() 函数：

```
// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void
freewalk(pagetable_t pagetable)
{
  // 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) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      freewalk((pagetable_t)child);
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      panic("freewalk: leaf");
    }
  }
  kfree((void*)pagetable);
}
```

freewalk() 函数通过递归调用来遍历所有的页表，并通过标志位来判断是否到达树的叶子结点。PTE2PA() 用于将逻辑地址转化为物理地址。

(2) **编写** vmprint() **函数**

仿照 freewalk() 函数可以编写出如下 vmprint() 函数:

```
void
_vmprint(pagetable_t pagetable, int level)
{
  for (int i = 0; i < 512; i++)
  {
    pte_t pte = pagetable[i];
    if (pte & PTE_V) // 页表项有效
    {
      uint64 child = PTE2PA(pte); // 转换到物理地址
      for (int j = 0; j < level; j++)
      {
        if(j)printf("␣");
        printf("..");
      }
      printf("%d:␣pte␣%p␣pa␣%p\n", i, pte, child);

      if((pte & (PTE_R|PTE_W|PTE_X)) == 0)
      {
        _vmprint((pagetable_t)child, level + 1);
      }
    }
  }

}

void
vmprint(pagetable_t pagetable)
{
  printf("page␣table␣%p\n", pagetable);
  _vmprint(pagetable, 1);
  return;
}
```

其中 level 用于记录到达第几层页表，以确定要输出几个..。

(3) **添加声明**

按照提示,在 kernel/exec.c 中的 return argc; 前添加 if(p->pid==1) vmprint(p->pagetable);:

```
if(p->pid==1) vmprint(p->pagetable);
return argc; // this ends up in a0, the first argument to main(argc, argv)
```

在在 kernel/def.hs 中添加 vmprint() 函数的声明，使得 exec.c 可以调用 vmprint() 函数:

```
// vm.c
...
void            vmprint(pagetable_t);
```

(4) **结果**

上述操作完成后可以得到以下结果：

```
make[1]: 离开目录"/home/ubuntu/桌面/xv6-labs-202
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (3.6s)
```

图 2: printout passed.

(5) **问题**

Explain the output of vmprint in terms of Fig 3-4 from the text. What does page 0 contain? What is in page 2? When running in user mode, could the process read/write the memory mapped by page 1?

图 3-4 是哪个？是指 xv6book 中的图么？如果是，page0 又指什么？

Part 3: A kernel page table per process

3.1 问题简述

本部分要求我们为每个进程新增一个内核态的页表，然后在该进程进入到内核态时，不使用公用的内核态页表，而是使用进程的内核态页表。

3.2 问题分析

3.3 实验步骤

(1) **添加内核页表字段**：

在 proc 结构体中增加一个字段，用于进程的内核页表。

```
// Per-process state
struct proc {
...
  pagetable_t kpagetable;       // Kernal page table
...
};
```

(2) **初始化内核页表**：

仿照 kernel/vm.c 里 kvmmap() 与 kvminit() 函数，重新写一个映射函数与一个初始化函数

```
void
ukvmmap(pagetable_t kpagetable,uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(kpagetable, va, sz, pa, perm) != 0)
    panic("uvmmap");
}

pagetable_t
ukvminit()
{
  pagetable_t kpagetable = (pagetable_t) kalloc();
  memset(kpagetable, 0, PGSIZE);
  ukvmmap(kpagetable,UART0, UART0, PGSIZE, PTE_R | PTE_W);
  ukvmmap(kpagetable,VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
  ukvmmap(kpagetable,CLINT, CLINT, 0x10000, PTE_R | PTE_W);
  ukvmmap(kpagetable,PLIC, PLIC, 0x400000, PTE_R | PTE_W);
  ukvmmap(kpagetable,KERNBASE, KERNBASE, (uint64)etext-KERNBASE,
  PTE_R | PTE_X);
  ukvmmap(kpagetable,(uint64)etext, (uint64)etext,
  PHYSTOP-(uint64)etext, PTE_R | PTE_W);
  ukvmmap(kpagetable,TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
  return kpagetable;
}
```

然后，在 allocproc 中调用该函数：

```
static struct proc*
allocproc(void)
{
...
  // An empty user page table.
  p->pagetable = proc_pagetable(p);
  if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  // 下面是新添加的内容
  // An empty user kernal page table.
  p->kpagetable = ukvminit(p);
  if(p->pagetable == 0){
```

```
    freeproc(p);
    release(&p->lock);
    return 0;
  }
...
}
```

并在 kernel/defs.h 添加函数声明：pagetable_t ukvminit(void); 以及 void ukvmmap(pagetable_t, uint64, uint64, uint64, int);

(3) **初始化内核栈**：

根据提示，接下来要将 procinit() 函数中部分或全部功能转移到 allocproc() 函数。首先观察 procinit() 函数：

```
// initialize the proc table at boot time.
void
procinit(void)
{
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");

      // Allocate a page for the process's kernel stack.
      // Map it high in memory, followed by an invalid
      // guard page.
      char *pa = kalloc();
      if(pa == 0)
        panic("kalloc");
      uint64 va = KSTACK((int) (p - proc));
      kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
      p->kstack = va;
   }
  kvminithart();
}
```

该函数是用于将每个进程的内核页表映射到一个针对该进程的内核堆栈。

于是转移到 allocproc() 函数，用上之前写好的 ukvmmap() 函数以初始化内核堆栈：

```
  // An empty user kernal page table.
  p->kpagetable = ukvminit(p);
  if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
```

```
    return 0;
  }

  // 初始化内核栈
  char *pa = kalloc();
      if(pa == 0)
        panic("kalloc");
      uint64 va = KSTACK((int) (p - proc));
      ukvmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
      p->kstack = va;
```

(4) **修改** scheduler() **函数，在进程调度时，切换内核页**：

按照提示，仿照 kvminithart() 函数，修改 scheduler() 函数将进程的内核页表加载到内核的 satp ，并且寄存器中在没有进程运行时使用 kernel_pagetable

```
void
scheduler(void)
{
 ...
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;

        // load the process' s kernel page table into the core' s satp
        // register
        w_satp(MAKE_SATP(p->kpagetable));
        sfence_vma();

        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
```

```
29
30          kvminithart(); // use kernel_pagetable when no process is running.
31  ...
32  }
```

(5) **在** freeproc() **函数中释放一个进程的内核页表**:

释放一个进程的内核页表首先要释放页表内的内核栈，因为页表内存储的内核栈地址本身就是一个虚拟地址，需要先将这个地址指向的物理地址进行释放:

```
1  if (p->kstack)
2  {
3  pte_t * pte = walk(p->kpagetable, p->kstack, 0);
4  kfree((void*)PA2PTE(*pte));
5  }
6  p->kstack = 0;
```

其中根据 xv6 book 介绍，walk() 函数模仿 RISC-V 分页硬件查找虚拟地址的 PTE。walk 每次降低 3 级页表的 9 位。它使用每一级的 9 位虚拟地址来查找下一级页表或最后一级的 PTE。

需要注意的是，这里需要在 kernel/defs.h 中添加 walk 函数的声明，否则无法直接引用。然后是释放页表，直接遍历所有的页表，释放所有有效的页表项即可。仿照 freewalk 函数。由于 freewalk 函数将对应的物理地址也直接释放了，我们这里释放的内核页表仅仅只是用户进程的一个备份，释放时仅释放页表的映射关系即可，不能将真实的物理地址也释放了。因此不能直接调用 freewalk 函数，而是需要进行更改:

```
1  void
2  proc_freewalk(pagetable_t pagetable)
3  {
4    for (int i = 0; i < 512; i++) {
5      pte_t pte = pagetable[i];
6          if (pte & PTE_V) {
7            pagetable[i] = 0;
8            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
9              uint64 child = PTE2PA(pte);
10                 proc_freewalk((pagetable_t)child);
11           }
12         }
13   }
14   kfree((void*)pagetable);
15 }
```

再在 freeproc 中进行调用:

```
// delete kernel pagetable
if(p->kpagetable) {
proc_freewalk(p->kpagetable);
}
p->kpagetable = 0;
```

(6) **切换进程的内核页表**:

最后，在 vm.c 中添加头文件:

```
#include "spinlock.h"
#include "proc.h"
```

然后更改 kvmpa 函数 (否则会报错: "panic:kvmpa"):

```
pte = walk(myproc()->kpagetable, va, 0);
```

```
== Test usertests ==
$ make qemu-gdb
(285.6s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
```

图 3: usertests passed

Part 4: Simplify copyin/copyinstr

4.1 问题简述

该部分主要目的是将用户进程页表的所有内容都复制到内核页表中.

4.2 实验步骤

(1) **替换** copyin(),copyinstr() **函数**:

根据提示，将 vm.c 中 copyin(),copyinstr() 函数的主体内容替换为已经写好的 vmcopyin.c 中的 copyin_new(),copyinstr_new() 函数:

```c
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  return copyin_new(pagetable,dst,srcva,len);
}

int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  return copyinstr_new(pagetable,dst,srcva,max);
}
```

并在 defs.h 中添加声明:

```c
// vmcopyin.c
int             copyin_new(pagetable_t, char *, uint64, uint64);
int             copyinstr_new(pagetable_t, char *, uint64, uint64);
```

(2) **修改** fork(),exec(),sbrk() **函数**:

在 fork() 函数中, 需要将父进程的 pagetable 复制给子进程的 kpagetable。而 fork() 函数中使用了 vm.c 中的 uvmcopy() 函数来将父进程的 pagetable 复制给子进程的 pagetable, 所以我们要仿照此函数, 写出一个函数 u2kvmcopy() 用来将父进程的 pagetable 复制给子进程的 kpagetable。

```c
void
u2kvmcopy(pagetable_t pagetable, pagetable_t kpagetable, uint64 oldsz,
uint64 newsz)
{
  pte_t *pte_from, *pte_to;
  uint64 a, pa;
  uint flags;

  if (newsz < oldsz)
    return;
  oldsz = PGROUNDUP(oldsz);
  for (a = oldsz; a < newsz; a += PGSIZE)
  {
    if ((pte_from = walk(pagetable, a, 0)) == 0)
      panic("u2kvmcopy: pte should exist");
    if ((pte_to = walk(kpagetable, a, 1)) == 0)
    // copy the pte to the same address at kernel page table
      panic("u2kvmcopy: walk fails");
    pa = PTE2PA(*pte_from);
```

```
20      flags = (PTE_FLAGS(*pte_from) & (~PTE_U));
21      *pte_to = PA2PTE(pa) | flags;
22    }
23  }
```

这个函数与 uvmcopy() 函数不同的地方在于 PTE_U ，由第四条提示，一个设置了 PTE_U 的页不能在内核模式下被访问，所以将在 line14 中将这个标志位去掉。

然后修改 fork() 函数，实现上述目标：

```
1  int
2  fork(void)
3  {
4  ...
5    u2kvmcopy(np->pagetable, np->kpagetable, 0, np->sz);
6  ...
7  }
```

exec() 函数在 kernel/exec.c 中，在执行新的程序前，初始化之后，进行页表拷贝：

```
1  int
2  exec(char *path, char **argv)
3  {
4  ...
5    // Load program into memory.
6    for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
7  ...
8      if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
9        goto bad;
10     // 添加检测，防止程序大小超过 PLIC
11     if(sz1 >= PLIC)
12       goto bad;
13     sz = sz1;
14 ...
15   }
16   iunlockput(ip);
17 ...
18   stackbase = sp - PGSIZE;
19
20   u2kvmcopy(pagetable, p->kpagetable, 0, sz);
21   // Push argument strings, prepare rest of stack in ustack.
22   for(argc = 0; argv[argc]; argc++) {
23 ...
24 }
```

sbrk() 函数是系统调用，位于 sysproc.c 中，具体内容如下：

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

可见其调用了 growproc() 函数，下面对于该函数进行修改。

```
int
growproc(int n)
{
  uint sz;
  struct proc *p = myproc();

  sz = p->sz;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
      return -1;
    }
    // 添加
    u2kvmcopy(p->pagetable, p->kpagetable, sz-n, sz);
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }
  p->sz = sz;
  return 0;
}
```

最后，在 userinit 中把第一个进程的用户页表包含在它的内核页表中。

```
void
userinit(void)
{
  struct proc *p;

```

```
  p = allocproc();
  initproc = p;

  // allocate one user page and copy init's instructions
  // and data into it.
  uvminit(p->pagetable, initcode, sizeof(initcode));
  p->sz = PGSIZE;
  // 添加
  u2kvmcopy(p->pagetable, p->kpagetable, 0, p->sz);

  // prepare for the very first "return" from kernel to user.
  p->trapframe->epc = 0;      // user program counter
  p->trapframe->sp = PGSIZE;  // user stack pointer

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&p->lock);
}
```

Part5 Question

1、Explain why the third test srcva + len < srcva is necessary in copyin_new(): give values for srcva and len for which the first two test fail (i.e., they will not cause to return -1) but for which the third one is true (resulting in returning -1).

实际上这样操作就是为了防止溢出。xv6book 中有提到

For example if(ph.vaddr + ph.memsz < ph.vaddr) checks for whether the sum overflows a 64-bit integer.

例如：

```
#include <iostream>
using namespace std;
int main()
{
    uint64_t srcva = UINT64_MAX / 2;
    uint64_t len = UINT64_MAX - srcva + 1;
    cout << srcva + len << " " << srcva << " " << len << endl;
    return 0;
}
```

输出结果

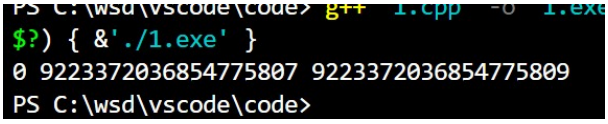

图 4: Result

Part 6: 实验结果

运行 make grade，检验结果：

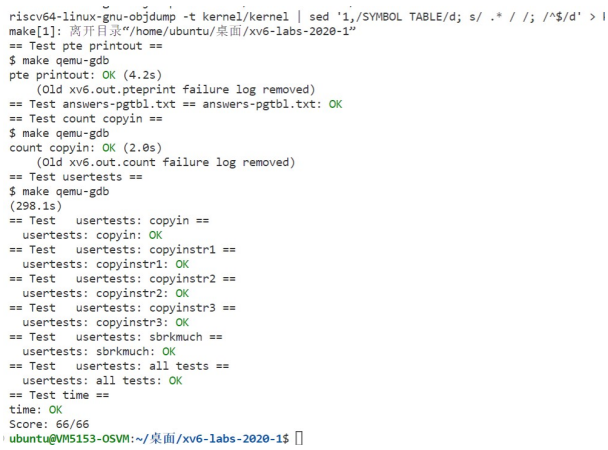

图 5: Result

本次实验完成了 Page Table 有关功能的实现和实际应用。在实验过程中得到了以下收获：

- 对于多级页表这个知识点有了深刻的认识
- 对于虚拟地址和物理地址的转换以及内存结构有了切身的感受
- 认识到了操作系统内存设计合理性以及正确性