

# 一、Lab4 实验报告

## 1.1. 打印页表

### 1.1.1 实验步骤

a. 首先实现vmprint\_rec函数用于递归调用 1. 函数参数包括pagetable、level(用于表示递归层数以判断缩进深度)、va(表示根据页表项索引而生成的虚拟地址) 2. 首先遍历512个页表项，如果与PTE\_V做与操作后为真，则表示该项有效，继续进行以下操作，否则跳过 3. 通过PTE2PA获取对应的虚拟地址，通过PTE\_FLAGS获取标志位，并根据level打印缩进 4. 根据freewalk函数， $(pte \& (PTE\_R \mid PTE\_W \mid PTE\_X)) \neq 0$ 表明该项为叶节点，所以根据要求打印即可 5. 否则根据要求打印后，需将va赋值为 $va \ll 9 \mid i$ (因为每层页表对应虚拟地址的9位)，而后继续递归

```
void vmprint_rec(pagetable_t pagetable, int level, uint64 va){
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if(pte & PTE_V){
            uint64 pa = PTE2PA(pte);
            uint64 flags = PTE_FLAGS(pte);
            printf("||");
            for(int j = 0; j < level; j++){
                printf("  ||");
            }

            if((pte & (PTE_R | PTE_W | PTE_X)) != 0){
                printf("idx: %d: va: %p -> pa: %p, flags: %s%s%s%s\n",
                    i,
                    (va << PXSHIFT(1) | i << PXSHIFT(0)),
                    pa,
                    (flags & PTE_R) ? "r" : "-",
                    (flags & PTE_W) ? "w" : "-",
                    (flags & PTE_X) ? "x" : "-",
                    (flags & PTE_U) ? "u" : "-");
            }else{
                printf("idx: %d: pa: %p, flags: %s%s%s%s\n",
                    i,
                    pa,
                    (flags & PTE_R) ? "r" : "-",
                    (flags & PTE_W) ? "w" : "-",
                    (flags & PTE_X) ? "x" : "-",
                    (flags & PTE_U) ? "u" : "-");
                va = va << 9 | i;
                vmprint_rec((pagetable_t)pa, level+1, va);
            }
        }
    }
}
```

```
}  
}  
}  
}
```

b. 而后实现vmprint函数，只需要打印要求的头部后调用vmprint\_rec进入递归打印即可

```
void vmprint(pagetable_t pagetable){  
    printf("page table %p\n", pagetable);  
    vmprint_rec(pagetable, 0, 0);  
}
```

c. 最后在exec函数中进行调用即可

```
if(p->pid == 1) vmprint(p->pagetable);
```

## 1.1.2 实验测试结果

```
== Test pte printout ==  
$ make qemu-gdb  
pte printout: OK (2.7s)
```

## 2. 独立内核页表

### 1.2.1 实验步骤

a. 首先在proc结构体中添加k\_pagetable和kstack\_pa分别用于记录独立内核页表以及内核栈物理地址，并实现indep\_kvminit函数用于初始化独立内核页表，该函数实现与kvminit相似，但无需映射CLINT

```
void indep_kvmmmap(pagetable_t pagetable, uint64 va, uint64 pa, uint64 sz, int perm)  
{  
    if (mappages(pagetable, va, sz, pa, perm) != 0) panic("kvmmmap");  
}  
  
pagetable_t indep_kvminit(){  
    pagetable_t indep_kernel_pagetable = (pagetable_t) kalloc();  
    memset(indep_kernel_pagetable, 0, PGSIZE);  
}
```

```

// uart registers
indep_kvmmmap(indep_kernel_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);

// virtio mmio disk interface
indep_kvmmmap(indep_kernel_pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

// PLIC
indep_kvmmmap(indep_kernel_pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

// map kernel text executable and read-only.
indep_kvmmmap(indep_kernel_pagetable, KERNBASE, KERNBASE, (uint64)etext -
KERNBASE, PTE_R | PTE_X);

// map kernel data and the physical RAM we'll make use of.
indep_kvmmmap(indep_kernel_pagetable, (uint64)etext, (uint64)etext, PHYSTOP -
(uint64)etext, PTE_R | PTE_W);

// map the trampoline for trap entry/exit to
// the highest virtual address in the kernel.
indep_kvmmmap(indep_kernel_pagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE,
PTE_R | PTE_X);

return indep_kernel_pagetable;
}

```

b. 而后修改procinit函数，由于此时使用独立内核页表，所以无需将va赋值给kstack，而是在后续allocproc时赋值，并需要记录内核栈物理地址到kstack\_pa用于allocproc中在独立内核页表进行映射。

```

// 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));
        kvmmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
        p->kstack_pa = (uint64)pa;
    }
    kvmminithart();
}

```

c. 然后修改allocproc函数，加入分配独立内核页表以及映射内核栈的过程

```

p->k_pagetable = indep_kvminit();
uint64 va = KSTACK((int)(0));
indep_kvmmap(p->k_pagetable, va, (uint64)p->kstack_pa, PGSIZE, PTE_R | PTE_W);
p->kstack = va;

```

d. 在scheduler函数中，由于此时使用独立内核页表，所以在进入选定进程时还需要进行一次页表切换以进入该进程的独立内核页表，并在退出时切换回全局内核页表

```

w_satp(MAKE_SATP(p->k_pagetable));
sfence_vma();

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

kvminithart();

```

e. 还需要修改freeproc函数以在进程结束时释放对应的独立内核页表，但是由于内核栈物理地址是在boot时分配的固定地址，所以无需释放。此时需要实现一个free\_indep\_kernel\_pagetable对独立内核页表进行释放，但不释放对应的实际物理地址，该函数仿照freewalk函数进行实现

```

void free_indep_kernel_pagetable(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);
            free_indep_kernel_pagetable((pagetable_t)child);
            pagetable[i] = 0;
        }
    }
    kfree((void *)pagetable);
}

static void freeproc(struct proc *p) {
    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->kstack = 0;
    if (p->k_pagetable) free_indep_kernel_pagetable(p->k_pagetable);
}

```

```
p->k_pagetable = 0;
p->state = UNUSED;
}
```

f. 在完成上述操作后，由于kvmpa函数中提供转换虚拟内核地址为物理地址的过程，所以还需修改其实现为访问独立内核页表而非全局内核页表。并且由于kernel/virtio\_disk.c中的virtio\_disk\_rw中使用了该函数，所以还需要对其中使用该函数处进行简单修改

```
uint64 kvmpa(pagetable_t pagetable, uint64 va) {
    uint64 off = va % PGSIZE;
    pte_t *pte;
    uint64 pa;

    pte = walk(pagetable, va, 0);
    if (pte == 0) panic("kvmpa");
    if ((*pte & PTE_V) == 0) panic("kvmpa");
    pa = PTE2PA(*pte);
    return pa + off;
}
```

```
disk.desc[idx[0]].addr = (uint64)kvmpa(myproc()->k_pagetable, (uint64)&buf0);
```

## 1.2.2 实验测试结果

```
$ kvmtest
kvmtest: start
test_pagetable: 1
kvmtest: OK
```

```
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3235
                sepc=0x0000000000005476 stval=0x0000000000005476
usertrap(): unexpected scause 0x000000000000000c pid=3236
                sepc=0x0000000000005476 stval=0x0000000000005476
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
```



```
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6215
    sepc=0x00000000000002048 stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6216
    sepc=0x00000000000002048 stval=0x000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6217
    sepc=0x00000000000002048 stval=0x00000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6218
    sepc=0x00000000000002048 stval=0x00000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6219
    sepc=0x00000000000002048 stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6220
    sepc=0x00000000000002048 stval=0x000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6221
    sepc=0x00000000000002048 stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6222
    sepc=0x00000000000002048 stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6223
    sepc=0x00000000000002048 stval=0x0000000080061a80
usertrap(): unexpected scause 0x000000000000000d pid=6224
    sepc=0x00000000000002048 stval=0x000000008006dd0
usertrap(): unexpected scause 0x000000000000000d pid=6225
    sepc=0x00000000000002048 stval=0x000000008007a120
usertrap(): unexpected scause 0x000000000000000d pid=6226
    sepc=0x00000000000002048 stval=0x0000000080086470
usertrap(): unexpected scause 0x000000000000000d pid=6227
    sepc=0x00000000000002048 stval=0x00000000800927c0
usertrap(): unexpected scause 0x000000000000000d pid=6228
    sepc=0x00000000000002048 stval=0x000000008009eb10
usertrap(): unexpected scause 0x000000000000000d pid=6229
    sepc=0x00000000000002048 stval=0x00000000800aae60
usertrap(): unexpected scause 0x000000000000000d pid=6230
    sepc=0x00000000000002048 stval=0x00000000800b71b0
usertrap(): unexpected scause 0x000000000000000d pid=6231
    sepc=0x00000000000002048 stval=0x00000000800c3500
usertrap(): unexpected scause 0x000000000000000d pid=6232
    sepc=0x00000000000002048 stval=0x00000000800cf850
usertrap(): unexpected scause 0x000000000000000d pid=6233
    sepc=0x00000000000002048 stval=0x00000000800dbba0
usertrap(): unexpected scause 0x000000000000000d pid=6234
    sepc=0x00000000000002048 stval=0x00000000800e7ef0
usertrap(): unexpected scause 0x000000000000000d pid=6235
    sepc=0x00000000000002048 stval=0x00000000800f4240
usertrap(): unexpected scause 0x000000000000000d pid=6236
    sepc=0x00000000000002048 stval=0x0000000080100590
usertrap(): unexpected scause 0x000000000000000d pid=6237
    sepc=0x00000000000002048 stval=0x000000008010c8e0
```

```
usertrap(): unexpected scause 0x000000000000000d pid=6238
```

```
usertrap(): unexpected scause 0x000000000000000d pid=6246
      sepc=0x0000000000002048 stval=0x000000008017a6b0
usertrap(): unexpected scause 0x000000000000000d pid=6247
      sepc=0x0000000000002048 stval=0x0000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6248
      sepc=0x0000000000002048 stval=0x0000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6249
      sepc=0x0000000000002048 stval=0x000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6250
      sepc=0x0000000000002048 stval=0x00000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6251
      sepc=0x0000000000002048 stval=0x00000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6252
      sepc=0x0000000000002048 stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6253
      sepc=0x0000000000002048 stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6254
      sepc=0x0000000000002048 stval=0x00000000801dc130
```

OK

```
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6266
      sepc=0x0000000000003ec2 stval=0x0000000000012000
```

OK

test sbrkarg: OK

test validatetest: OK

```
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6270
      sepc=0x00000000000021b6 stval=0x000000000000fbc0
```

OK

test opentest: OK

test writetest: OK

test writebig: OK

test createtest: OK

test openiput: OK

test exitiput: OK

test iput: OK

test mem: OK

test pipe1: OK

test preempt: kill... wait... OK

test exitwait: OK

test rmdot: OK

test fourteen: OK

test bigfile: OK

test dirfile: OK

test iref: OK

test forktest: OK

test bigdir: OK

ALL TESTS PASSED

## 二、问题解答



## 2.1

---

- 在寻找物理地址时，首先根据该虚拟地址的38-30位(1FE)在L2页表中(其地址被填充在satp寄存器中)查找相应的页表项得到L1页表的物理页帧号(页表项的10-53位)，将其0-11位的偏移量置为0得到对应的物理地址
- 而后在上述得到的L1页表中根据该虚拟地址的29-21位(F1)在L1页表中查找相应的页表项得到L0页表的物理页帧号，同样将其0-11位的偏移量置为0得到对应的物理地址
- 最后在上述得到的L0页表中根据该虚拟地址的20-12位(6A)在L0页表中查找相应的页表项得到该虚拟地址对应的物理页帧号(共44位)
- 此时根据得到的物理页帧以及虚拟地址中未使用的0-11位作为偏移进行拼接便可以得到实际的物理地址

## 2.2 问题2内容

---

由于每级页表的大小需要和页的大小相同，即4KB。每个页表项占用8个字节，因此每个页表中可以容纳的页表项数量为  $\frac{4\text{KB}}{8\text{Byte}} = 512$  项。又因为  $2^9 = 512$ ，所以索引只能是9位以满足页表的大小要求。