

# Lab4 实验报告

## 一、实验内容

### 1. 任务一：打印页表

#### 1.1 实验目标

在 exec 中插桩一个打印函数，使得 xv6 启动时会打印首个进程的页表信息。

- 在 `kernel/vm.c` 中实现 `vmprint()`
- 在 `exec()` 函数中插入语句 `if(p->pid==1) vmprint(p->pagetable)`，这条语句插在 `exec.c` 中 `return argc` 代码之前，即在第一个进程启动时打印页表信息。
- 在 `kernel/defs.h` 中定义 `vmprint()` 的接口

#### 1.2 实验步骤

##### 1.2.1 实现 `vmprint()`

参考 `freewalk()` 的遍历页表的过程：

```
1 // Recursively free page-table pages.
2 // All leaf mappings must already have been removed.
3 void freewalk(pagetable_t pagetable) {
4     // there are 2^9 = 512 PTEs in a page table.
5     // 遍历一个页表页的PTE表项
6     for (int i = 0; i < 512; i++) {
7         pte_t pte = pagetable[i]; // 获取第i条PTE
8         /* 判断PTE的Flag位, 如果还有下一级页表(即当前是根页表或次页表),
9            则递归调用freewalk释放页表项, 并将对应的PTE清零 */
10        if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
11            // this PTE points to a lower-level page table.
12            uint64 child = PTE2PA(pte); // 将PTE转为为物理地址
13            freewalk((pagetable_t)child); // 递归调用freewalk
14            pagetable[i] = 0; // 清零
15        } else if (pte & PTE_V) {
16            /* 如果叶子页表的虚拟地址还有映射到物理地址, 报错panic.
17               因为调用freewalk之前应该会先uvmunmap释放物理内存 */
18            panic("freewalk: leaf");
19        }
20    }
```



```

14         printf("  ||");
15     }
16     // 打印当前索引和页表项信息
17     if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
18         // 非叶子节点
19         printf("idx: %d: pa: %p, flags: ----\n", i, PTE2PA(pte));
20         _vmprint((pagetable_t)PTE2PA(pte), level + 1); // 递归调用
21     } else {
22         // 叶子节点 打印虚拟地址va
23         // uint64 va = ((uint64)pagetable) + (i << 12);
24         uint64 va = (i << 12); // 计算虚拟地址
25         printf("idx: %d: va: %p -> pa: %p, flags: %s%s%s%s\n", i,
26             va, PTE2PA(pte),
27             (pte & PTE_R ? "r" : "-"), (pte & PTE_W ? "w" : "-"),
28             (pte & PTE_X ? "x" : "-"), (pte & PTE_U ? "u" : "-"));
29     }
30 }
31 }
32 }
33
34 // 外部调用的 vmprint 函数
35 void vmprint(pagetable_t pagetable) {
36     // 打印根页表
37     printf("page table %p\n", pagetable);
38     // 传递level级和递归
39     _vmprint(pagetable, 1);
40 }

```

### 1.2.2 修改 exec.c/exec()

- 在 `exec()` 函数中插入语句 `if(p->pid==1) vmprint(p->pagetable)`，这条语句插在 `exec.c` 中 `return argc` 代码之前，即在第一个进程启动时打印页表信息。

```

1 int exec(char *path, char **argv) {
2     ...
3     proc_freepagetable(oldpagetable, oldsz);
4
5     if(p->pid==1) vmprint(p->pagetable); //调用vmprint
6     return argc; // this ends up in a0, the first argument to main(argc, argv)
7
8 bad:
9     if (pagetable) proc_freepagetable(pagetable, sz);
10    if (ip) {
11        iunlockput(ip);
12        end_op();

```

```

13     }
14     if(p->pid==1) vmprint(p->pagetable); //调用vmprint
15     return -1;
16 }

```

- 在 `kernel/defs.h` 中定义 `vmprint()` 的接口

```

1 void          vmprint(pagetable_t);

```

## 1.3 实验结果

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f24000
||idx: 0: pa: 0x0000000087f20000, flags: ----
||  ||idx: 0: pa: 0x0000000087f1f000, flags: ----
||  ||  ||idx: 0: va: 0x0000000000000000 -> pa: 0x0000000087f21000, flags: rwxu
||  ||  ||idx: 1: va: 0x0000000000001000 -> pa: 0x0000000087f1e000, flags: rwx-
||  ||  ||idx: 2: va: 0x0000000000002000 -> pa: 0x0000000087f1d000, flags: rwxu
||idx: 255: pa: 0x0000000087f23000, flags: ----
||  ||idx: 511: pa: 0x0000000087f22000, flags: ----
||  ||  ||idx: 510: va: 0x00000000001fe000 -> pa: 0x0000000087f76000, flags: rw--
||  ||  ||idx: 511: va: 0x00000000001ff000 -> pa: 0x0000000080007000, flags: r-x-
init: starting sh

```

```

== Test pte printout ==
$ make qemu-gdb
pte printout: OK (7.1s)
(Old xv6.out.ptepprint failure log removed)

```

## 2. 任务二：独立内核页表

### 2.1 实验目标

- 共享内核页表的映射：虚实地址相同，也就是直接映射。
- 独立内核页表的映射：虚实地址相同的映射应该要保留。

### 2.2 实验步骤

#### 2.2.1 修改 `kernel/proc.h` 中的 `struct proc`

```

1 // TODO
2 pagetable_t k_pagetable; // kernel page table

```

## 2.2.2 仿照 `kvminit()` 函数重新写一个创建内核页表的函数。

在vm.c中添加函数 `proc_kpt_init()`

```

1 // 为进程的内核页表新建一个初始化函数
2 pagetable_t proc_kpt_init(){
3
4     pagetable_t k_pagetable = (pagetable_t) kalloc();
5     memset(k_pagetable, 0, PGSIZE);
6
7     // uart registers
8     proc_kvmmap(k_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
9
10    // virtio mmio disk interface
11    proc_kvmmap(k_pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
12
13    // CLINT
14    proc_kvmmap(k_pagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
15
16    // PLIC
17    proc_kvmmap(k_pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
18
19    // map kernel text executable and read-only.
20    proc_kvmmap(k_pagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R
| PTE_X);
21
22    // map kernel data and the physical RAM we'll make use of.
23    proc_kvmmap(k_pagetable, (uint64)etext, (uint64)etext, PHYSTOP-
(uint64)etext, PTE_R | PTE_W);
24
25    // map the trampoline for trap entry/exit to
26    // the highest virtual address in the kernel.
27    proc_kvmmap(k_pagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R |
PTE_X);
28    return k_pagetable;
29 }
30
31 // kvmmap是为内核页表的虚拟地址与物理地址做映射，这里需要重新添加一个类似的函数
32 void proc_kvmmap(pagetable_t k_pagetable, uint64 va, uint64 pa, uint64 sz, int
perm){
33     if(mappages(k_pagetable, va, sz, pa, perm) != 0)
34         panic("proc_kvmmap");

```

### 2.2.3 修改 allocproc 函数

- 在 allocproc 中调用 proc\_kpt\_init() 函数
  - p->k\_pagetable = proc\_kpt\_init();
- 仿照 /kernel/proc.c 文件中的 procinit 函数中初始化了每个进程的内核栈，在 allocproc 函数中也仿照。

```

1  // An empty kernel page table.
2  p->k_pagetable = proc_kpt_init();
3  // 申请内核栈，确保每一个进程的内核页表都关于该进程的内核栈有一个映射
4  char *pa = kalloc();
5  if(pa == 0)
6      panic("kalloc");
7  uint64 va = KSTACK((int) (p - proc));
8  proc_kvmmmap(p->k_pagetable, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
9  p->kstack = va;

```

### 2.2.4 添加 proc\_kvminithart 函数

参照 kvminithart 函数，在其附近新添加 proc\_kvminithart 函数，以实现传递页表指针。

```

1 void
2 proc_kvminithart(pagetable_t k_pagetable){
3     w_satp(MAKE_SATP(k_pagetable));
4     sfence_vma();
5 }

```

### 2.2.5 修改 scheduler() 函数

在 scheduler() 函数中，进程切换前调用 proc\_kvminithart() 函数。

```

1     c->proc = p;
2     // 加载进程的内核页表到核心的satp寄存器
3     proc_kvminithart(p->k_pagetable);
4     switch(&c->context, &p->context);
5     // ljb add Come back to the global kernel page table
6     kvminithart();

```

```

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

```

## 2.2.6 添加 free\_proc\_kpt() 函数

参照 freewalk 函数在 vm.c 文件中添加 free\_proc\_kpt () 函数

```

1  // 释放进程的内核页表
2  void
3  free_proc_kpt(pagetable_t pagetable)
4  {
5      // there are 2^9 = 512 PTEs in a page table.
6      for(int i = 0; i < 512; i++){
7          pte_t pte = pagetable[i];
8          if(pte & PTE_V){
9              // this PTE points to a lower-level page table.
10             uint64 child = PTE2PA(pte);
11             pagetable[i] = 0;
12             if((pte & (PTE_R|PTE_W|PTE_X)) == 0){// 说明不是第三级，进行递归
13                 free_proc_kpt((pagetable_t)child);
14             }
15         }
16     }
17     kfree((void*)pagetable);
18 }
19

```

## 2.2.7 修改 proc.c/freeproc ()

在 freeproc () 函数中，释放内核栈和内核页表

```

1  static void freeproc(struct proc *p) {
2      ...
3      p->pagetable = 0;
4      // 释放一个进程的内核栈
5      if(p->kstack){
6          uvmunmap(p->k_pagetable, p->kstack, 1, 1);
7      }
8      p->kstack = 0;
9      // 释放内核页表
10     free_proc_kpt(p->k_pagetable);

```

```
11     p->k_pagetable = 0;
12     ...
13 }
```

### 2.2.8 在 `defs.h` 文件中添加以上函数的声明

```
1 pagetable_t    proc_kpt_init();
2 void           proc_kvmmmap(pagetable_t, uint64 , uint64 , uint64 , int );
3 void           proc_kvminithart(pagetable_t );
4 void           free_proc_kpt(pagetable_t pagetable);
```

### 2.2.9 修改 `kvmpa`

```
1 uint64 kvmpa(uint64 va) {
2     uint64 off = va % PGSIZE;
3     pte_t *pte;
4     uint64 pa;
5
6     pte = walk(myproc()->k_pagetable, va, 0); //修改
7     if (pte == 0) panic("kvmpa");
8     if ((*pte & PTE_V) == 0) panic("kvmpa");
9     pa = PTE2PA(*pte);
10    return pa + off;
11 }
```

## 2.3 实验结果

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



```

== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (1.5s)
== Test usertests ==
$ make qemu-gdb
(259.7s)
== 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
Score: 90/100

```

## 二、问题解答

### 问题一

1. 阅读参考材料 [xv6 book Chapter 3 Page tables](#) 以及 [xv6中的虚拟内存管理Links to an external site.](#)，阐述SV39标准下，给定一个64位虚拟地址为0xFFFFFE789ABCDEF的时候，是如何一步一步得到最终的物理地址的？（页表内容可以自行假设）

虚拟地址和物理地址映射图如下所示：

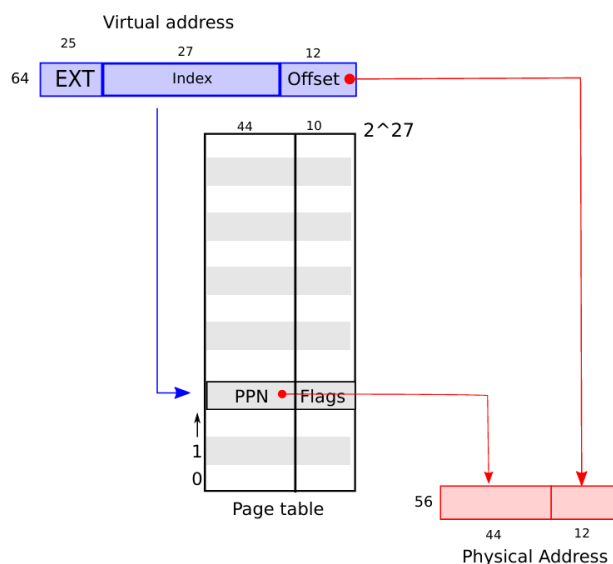


Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table.

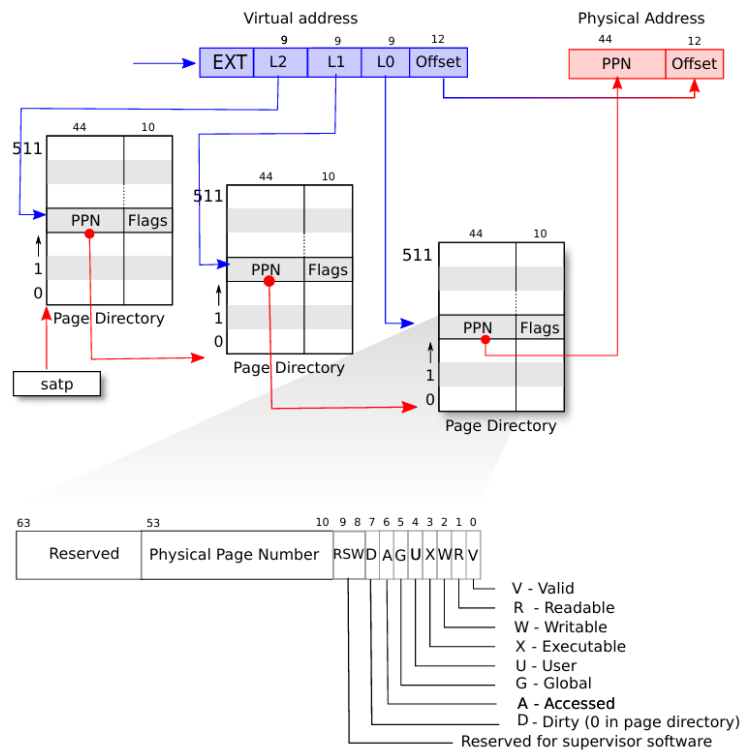


Figure 3.2: RISC-V address translation details.

## 步骤1：分割虚拟地址

给定的 64 位虚拟地址为 `0xFFFFF789ABCDEF`。

转化为二进制为：

```
1111 1111 1111 1111 1111 1111 1110 0111 1000 1001 1010 1011 1100 1101
1110 1111
```

- L2 (9位) : `110 0111 10` (从虚拟地址的第30位到第38位)
- L1 (9位) : `00 1001 101` (从虚拟地址的第21位到第29位)
- L0 (9位) : `0 1011 1100` (从虚拟地址的第12位到第20位)
- Offset (12位) : `0xDEF` ( `1101 1110 1111` ) (虚拟地址的低12位)

## 步骤2：查找根页表

根页表的物理地址存储在 `satp` 寄存器中

使用 L2 作为索引，在根页表中查找对应的次页表的物理地址：

- 假设查找到的 PPN 为 `0x200000000000` (44位)

## 步骤3：查找次页表

使用步骤 2 中得到的物理地址作为次页表的基地址，使用 L1 作为索引，在次页表中查找对应的叶子页表的物理地址：

- 假设查找到的 PPN 为 `0x300000000000`

## 步骤4：查找叶子页表

使用步骤 3 中得到的物理地址作为叶子页表的基地址，使用 L0 作为索引，在叶子页表中查找对应的物理页帧号（PPN）：

- 假设查找到的 PPN 为 `0x400000000000`

## 步骤5：构造物理地址

最后，我们将物理页帧号（PPN）与页内偏移量（Offset）组合起来，形成最终的物理地址：

- 物理页帧号： `0x4000 0000 000`
- 物理地址： `(0x400000000000 << 12) | 0xDEF`

## 结果

最终的物理地址为：

`(0x400000000000 << 12) | 0xDEF = 0x400000000000DEF`

因此，虚拟地址 `0xFFFFFE789ABCDEF` 在假设的页表项和页表结构下，被翻译为物理地址 `0x400000000000DEF`。

## 问题二

2. 我们注意到，SV39标准下虚拟地址的L2, L1, L0 均为9位。这实际上是设计中的必然结果，它们只能是9位，不能是10位或者是8位，你能说出其中的理由吗？（提示：一个页目录的大小必须与页的大小等大）

SV39 标准下，虚拟地址的 L2, L1, L0 均为 9 位，这是因为：

1. 一个页的大小是 4KB，可以容纳 512 个 8 字节的页目录项。
2. 512 个项需要 9 位来表示索引范围（从 0 到 511）。

3. 页目录项（PTE）必须填充整个页，以保持内存对齐，这样可以提高内存访问的效率。因此要确保了页目录的大小与页的大小等大，即每个页表（根页表、次页表、叶子页表）都是一个完整的页，这样可以高效地利用内存，并保持一致的页表结构。所以它们只能是9位。