浙江大学

本科实验报告

课程名称：操作系统

姓　　名：夏尤楷

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3210104331

指导教师：夏莹杰

2023 年 11 月 26 日

浙江大学操作系统实验报告

实验名称：RV64 虚拟内存管理

电子邮件地址：459510812@qq.com 手机：15058004449

实验地点：玉泉曹光彪西楼 503 实验日期：2023 年 11 月 26 日

一、实验目的和要求

1. 学习虚拟内存的相关知识，实现物理地址到虚拟地址的切换。
2. 了解 RISC-V 架构中 SV39 分页模式，实现虚拟地址到物理地址的映射，并对不同的段进行相应的权限设置。

二、实验过程

（一） 准备工程

在 defs.h 添加如下内容：

```
#define OPENSBI_SIZE (0x200000)

#define VM_START (0xffffffe000000000)
#define VM_END   (0xffffffff00000000)
#define VM_SIZE  (VM_END - VM_START)

#define PA2VA_OFFSET (VM_START - PHY_START)
```

从 repo 同步 vmlinux.lds，替换 arch/riscv/kernel 目录下原有的 vmlinux.lds。

从本实验开始我们需要使用刷新缓存的指令扩展，并自动在编译项目前执行 clean 任务来防止对头文件的修改无法触发编译任务。因此，对顶层目录的 Makefile 作以下修改：（蓝色字即为出现修改的部分）

```
...
ISA=rv64imafd_zifencei
```

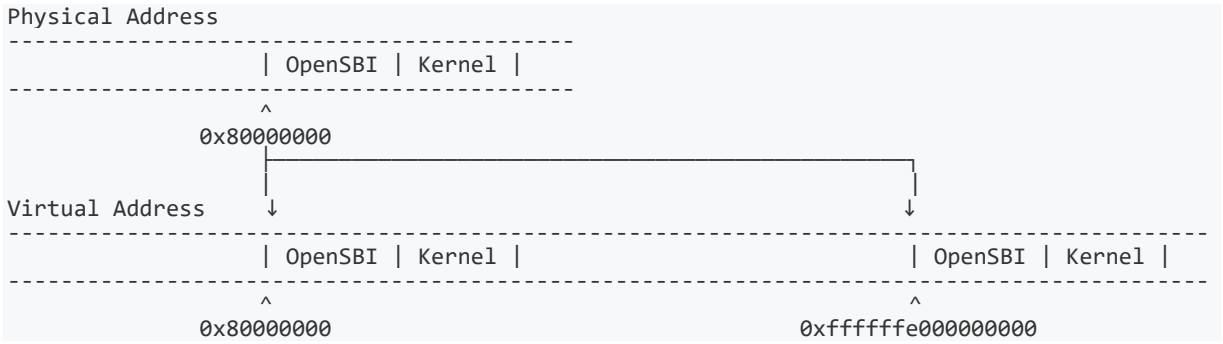
```
3.  ...
4.  all: clean
5.      ${MAKE} -C lib all
6.      ${MAKE} -C test all
7.      ${MAKE} -C init all
8.      ${MAKE} -C arch/riscv all
9.      @echo -e '\n'Build Finished OK
10. ...
```

（二） 开启虚拟内存映射、

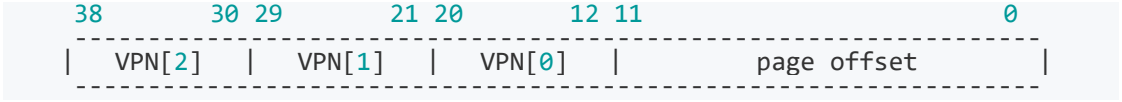
在 RISC-V 中开启虚拟地址被分为了两步：setup_vm 以及 setup_vm_final。在 arch/riscv/kernel 目录下创建 vm.c 源文件，将每一步分别通过同名函数进行实现。

1. setup_vm 的实现

将 0x80000000 开始的 1GB 区域进行两次映射，其中一次是等值映射（PA == VA），另一次是将其映射至高地址（PA + PV2VA_OFFSET == VA）。如下图所示：

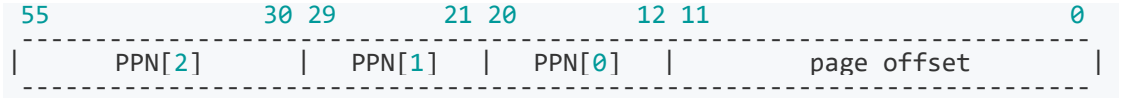


其中，映射是指将虚拟内存页的地址与物理内存页的地址之间建立对应关系，这种对应关系保存在页表中。本实验使用 Sv39 型地址。Sv39 型虚拟地址有 64 位，格式如下：



可见虚拟地址的 64 位只有低 39 位有效。

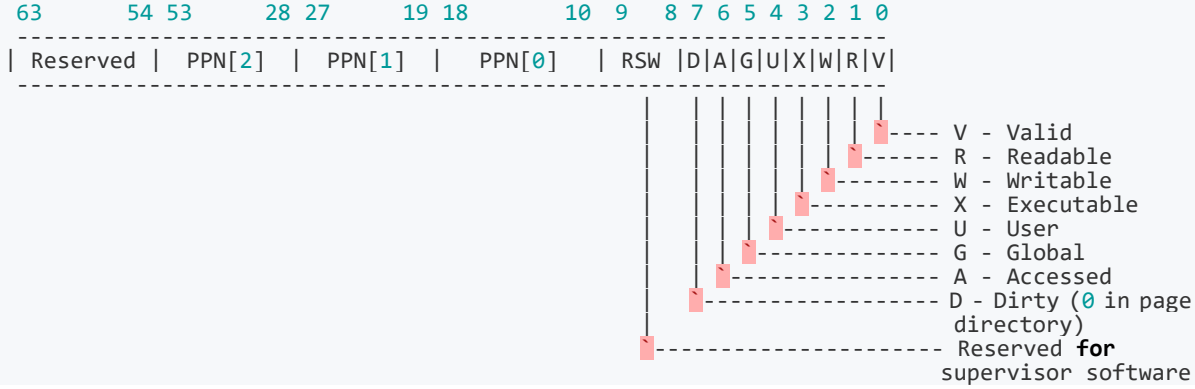
Sv39 型物理地址有 56 位，格式如下：



Sv39 支持三级页表结构，VPN[2-0]（Virtual Page Number）分别代表每级页表的虚拟页号，PPN[2-0]（Physical Page Number）分别代表每级页表的物理页号。1 个物理页大小为 4KB，物理页号 = 物理地址 >> 12。物理地址和虚拟地址的低 12 位表示页内

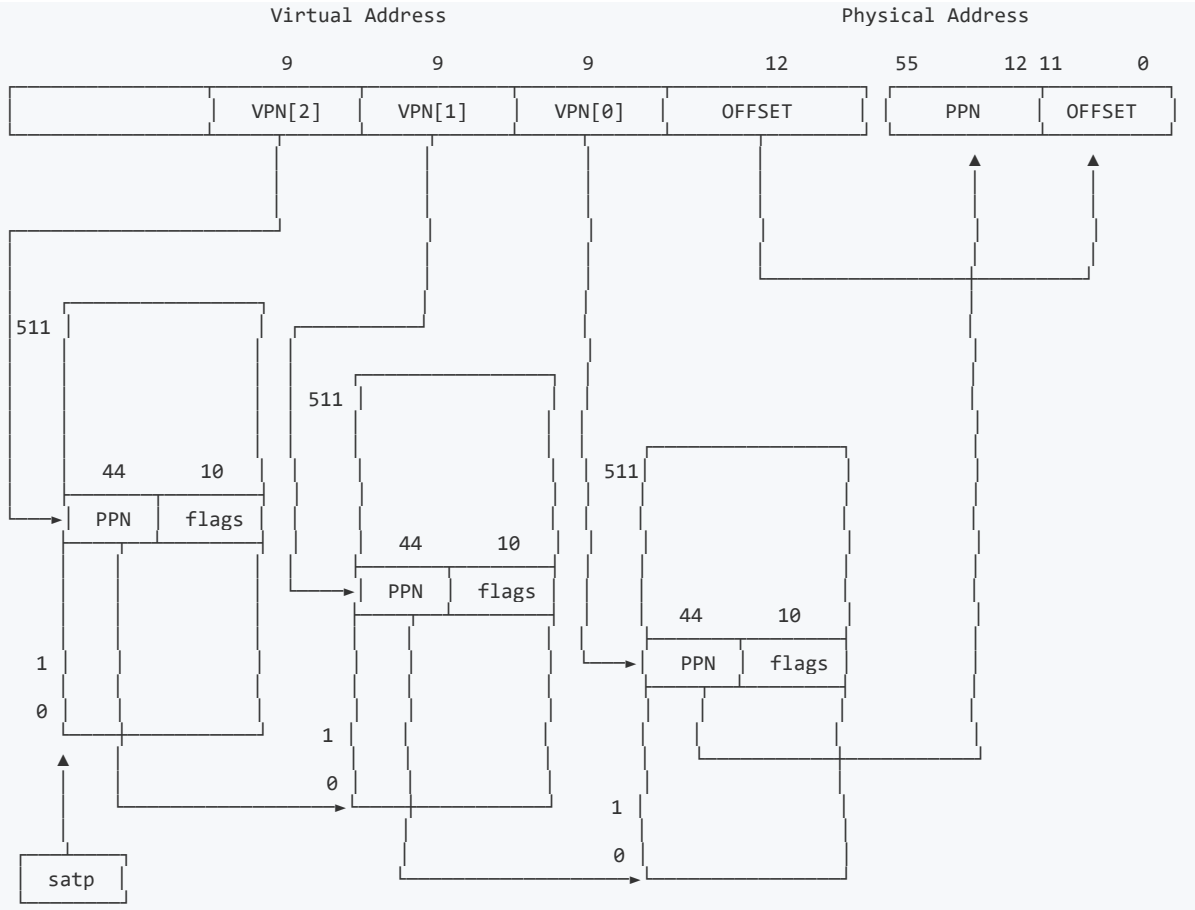
偏移（page offset）。

页表中会存储页表条目（Page Table Entry，PTE），每个条目的格式如下：



其中，0～9 位是保护位。V 为有效位，当 V = 0, 访问该 PTE 会产生 Pagefault；R = 1 时该页可读；W = 1 时该页可写；X = 1 时该页可执行；U、G、A、D、RSW 本次实验中设置为 0 即可。

通过页表条目将虚拟地址转化为物理地址的流程图如下（以三级页表为例）：



综合以上原理，将 setup_vm 实现如下：

```c
unsigned long  early_pgtbl[512] __attribute__((__aligned__(0x1000)));

void setup_vm(void) {
    /*
    1. 由于是进行 1GB 的映射，这里不需要使用多级页表。
    2. 将 va 的 64bit 作如下划分： | high bit | 9 bit | 30 bit |
        high bit 可以忽略；
        中间 9 bit 作为 early_pgtbl 的 index；
        低 30 bit 作为页内偏移。这里注意到 30 = 9 + 9 + 12，即我们只使用根页表，根页表的每个 entry 都对应 1GB 的区域。
    3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    */
    memset(early_pgtbl,0x0,PGSIZE);

    int index = PHY_START >> 30 & 0x1ff;

    early_pgtbl[index] = (PHY_START >> 30 & 0x3fff) << 28 | 15;
    index = VM_START >> 30 & 0x1ff;
    early_pgtbl[index] = (PHY_START >> 30 & 0x3fff) << 28 | 15;

    return;
}
```

在 head.S 的合适位置调用函数 setup_vm。

在执行完 setup_vm 后，再在 head.S 中调用 relocate 函数，完成对 satp 寄存器的设置，以及跳转到相应的物理地址。

其中，satp 寄存器的格式如下：

由于本实验使用 Sv39 地址，MODE 字段的值置为 8。ASID（Address Space Identifier）在此次实验中直接置 0 即可。PPN（Physical Page Number）为顶级页表的物理页号。

于是，在 head.S 中，relocate 函数实现如下：

```assembly
relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    .set PA2VA_OFFSET, 0xffffffdf80000000
    li t0, PA2VA_OFFSET
    add ra, ra, t0
    add sp, sp, t0

    # set satp with early_pgtbl
    la t2, early_pgtbl
    sub t2, t2, t0 # VA->PA
    srli t2, t2, 12 # PA->PPN
    # t0 = MODE8
    addi t0, x0, 1
    li t1, 63
    sll t0, t0, t1
    or t2, t2, t0
```

```
    csrw satp, t2

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret
```

至此，我们已经完成了虚拟地址的开启，之后我们运行的代码也都将在虚拟地址上运行。

2. setup_vm_final 的实现

由于 setup_vm_final 中需要申请页面的接口，应该在其之前完成内存管理初始化，需要修改 mm.c 中的代码，mm.c 中初始化的函数（即 mm_init）接收的起始结束地址需要调整为虚拟地址，修改后代码如下：

```
void mm_init(void) {
    kfreerange(_ekernel, (char *)(PHY_END + PA2VA_OFFSET));
    printk("...mm_init done!\n");
}
```

在 setup_vm_final 中，对所有物理内存（128MB）进行映射，并设置正确的权限。此时不再需要进行等值映射，并采用三级页表进行映射。代码如下：

```
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, uint64 perm);

unsigned long  swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
extern uint64 _skernel,_stext, _srodata, _sdata, _sbss;
void setup_vm_final(void) {
     memset(swapper_pg_dir, 0x0, PGSIZE);
    // No OpenSBI mapping required
    // mapping kernel text X|-|R|V
     create_mapping(swapper_pg_dir, (uint64)&_stext, (uint64)&_stext - PA2VA_OFFSET,
                    (uint64)&_srodata - (uint64)&_stext,11);
    // mapping kernel rodata -|-|R|V
     create_mapping(swapper_pg_dir, (uint64)&_srodata, (uint64)&_srodata - PA2VA_OFFSET,
                    (uint64)&_sdata - (uint64)&_srodata,3);
    // mapping other memory -|W|R|V
     create_mapping(swapper_pg_dir, (uint64)&_sdata, (uint64)&_sdata - PA2VA_OFFSET,
                     PHY_SIZE - ((uint64)&_sdata - (uint64)&_stext),7);

    // set satp with swapper_pg_dir
asm volatile (
        "mv t0, %[swapper_pg_dir]\n"

        ".set _pa2va_, 0xffffffdf80000000\n"
        "li t1, _pa2va_\n"
        "sub t0, t0, t1\n"
        "srli t0, t0, 12\n"
        "addi t2, zero, 1\n"
        "slli t2, t2, 31\n"
        "slli t2, t2, 31\n"
        "slli t2, t2, 1\n"
```

```
        "or t0, t0, t2\n"

        "csrw satp, t0\n"

        : : [swapper_pg_dir] "r" (swapper_pg_dir)
        : "memory"
    );
//      YOUR CODE HERE

    // flush TLB
    asm volatile("sfence.vma zero, zero");

    // flush icache
    asm volatile("fence.i");
    return;
}


/**** 创建多级页表映射关系 *****/
/* 不要修改该接口的参数和返回值 */
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, uint64 perm) {
    uint64 VPN[3];
    uint64 *page_table[3];
    uint64 *new_page;

    for (uint64 addr = va; addr < va + sz; addr += PGSIZE, pa += PGSIZE) {
        page_table[2] = pgtbl;

        // 为每个级别的页表计算VPN
        VPN[2] = (addr >> 30) & 0x1ff;
        VPN[1] = (addr >> 21) & 0x1ff;
        VPN[0] = (addr >> 12) & 0x1ff;

        // 检查并创建每个级别的页表项
        for (int level = 2; level > 0; level--) {
            if ((page_table[level][VPN[level]] & 1) == 0) {
                new_page = (uint64 *)kalloc();
                page_table[level][VPN[level]] = ((((uint64)new_page - PA2VA_OFFSET) >> 12) << 10) | 1;
            }
            page_table[level - 1] = (uint64 *)((page_table[level][VPN[level]] >> 10) << 12);
        }

        // 设置最后一级页表项
        page_table[0][VPN[0]] = (perm & 0b1111) | ((pa >> 12) << 10);
    }
}
```

其中，对于 text、rodata 和 other memory 这三个内存部分，应当设置的权限是不同的。为了获得这三个部分各自对应的地址范围，以准确地设置这三部分内存的权限，可以根据链接脚本 vmlinux.lds 中的_stext、_srodata、_sdata 这几个量来确定设置权限的地址范围，这也是为什么会有“extern uint64 _skernel,_stext, _srodata, _sdata, _sbss;”这一行外部变量声明的代码。

最后，对 head.S 作调整，调整完后的 head.S 代码如下：

```
.extern start_kernel

.set PA2VA_OFFSET, 0xffffffdf80000000
.set _shift_left_satp_mode8, 0x3f
    .section .text.init
    .globl _start
```

```
_start:
    la sp, boot_stack_top # store the address of the stack top into the register sp
    li t0, PA2VA_OFF
    sub sp, sp, t0


    call setup_vm
    call relocate
    call mm_init
    call setup_vm_final
    call task_init

    la t0,_traps
    csrw stvec,t0
    # set stvec = _traps

    csrr t0,sie
    ori t0,t0,0x20
    csrw sie,t0
    # set sie[STIE] = 1

    andi a7,x0,0x00
    andi a6,x0,0
    andi a5,x0,0
    andi a4,x0,0
    andi a3,x0,0
    andi a2,x0,0
    andi a1,x0,0
    li t0,10000000
    rdtime a0
    add a0,a0,t0
    ecall
    # set first time interrupt

    csrr t0,sstatus
    ori t0,t0,0x2
    csrw sstatus,t0
    # set sstatus[SIE] = 1

    jal start_kernel

relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    li t0, PA2VA_OFFSET
    add ra, ra, t0
    add sp, sp, t0

    # set satp with early_pgtbl
    la t2, early_pgtbl
    sub t2, t2, t0 # VA->PA
    srli t2, t2, 12 # PA->PPN
    # t0 = MODE8
    addi t0, x0, 1
    li t1, 63
    sll t0, t0, t1
    or t2, t2, t0
    csrw satp, t2
```

```
91.
92.     # flush tlb
93.     sfence.vma zero, zero
94.
95.     # flush icache
96.     fence.i
97.
98.     ret
99.
100.    .section .bss.stack
101.    .globl boot_stack
102. boot_stack:
103.    .space 4096 # <-- change to your stack size(4KB)
104.
105.    .globl boot_stack_top
106. boot_stack_top:
```

（三） 编译及测试

在 proc.c 的 dummy 函数中，将代码“printk("[PID = %d] is running. auto_inc_local_var = %d\n", current->pid, auto_inc_local_var);”改为“printk("[PID = %d] is running. thread space begin at = %lx\n", current->pid, current);”。这样，就可以查看每个正在运行的线程被分配到的虚拟内存的首地址。

程序运行结果如下图。

```
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...mm_init done!
48
56
64
...proc_init done!
2022 Hello RISC-V
[S] Supervisor mode time interrupt!

switch to [PID = 8 COUNTER = 1]
[PID = 8] is running. thread space begin at = ffffffe007fb6000
[S] Supervisor mode time interrupt!

switch to [PID = 17 COUNTER = 1]
[PID = 17] is running. thread space begin at = ffffffe007fad000
[S] Supervisor mode time interrupt!

switch to [PID = 18 COUNTER = 1]
[PID = 18] is running. thread space begin at = ffffffe007fac000
[S] Supervisor mode time interrupt!

switch to [PID = 7 COUNTER = 2]
[PID = 7] is running. thread space begin at = ffffffe007fb7000
[S] Supervisor mode time interrupt!
[PID = 7] is running. thread space begin at = ffffffe007fb7000
[S] Supervisor mode time interrupt!

switch to [PID = 14 COUNTER = 2]
[PID = 14] is running. thread space begin at = ffffffe007fb0000
[S] Supervisor mode time interrupt!
[PID = 14] is running. thread space begin at = ffffffe007fb0000
```

三、讨论和心得

这次实验让我对虚拟内存的工作原理有了更加深入的理解。虚拟内存的工作原理似乎还

是蛮复杂和抽象的。它和物理内存的关系，也就是基于页表的映射关系，我还没有完全搞明白。但总比实验前懂了一些。

四、思考题

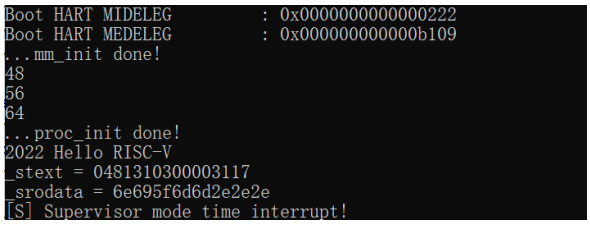
1. 指令和数据都能被正常使用，说明.text 段和.rodata 段所处的内存页都是有效的。

编译成功后能正确运行，说明.text 段具有执行权限，而.rodata 段不具有执行权限（要不然程序就会执行一些我们未编写的指令，出现奇怪的行为了）。

在 main.c 中，声明外部变量_stext、_srodata（类型为 uint64，分别代表.text 段和.rodata 段的数据），在函数 start_kernel 中加入以下两行代码，读取.text 段和.rodata 段内数据并在屏幕上输出：

```
printk("_stext = %lx\n", _stext);
printk("_srodata = %lx\n", _srodata);
```

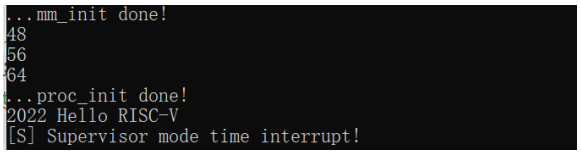
编译运行后，发现数据能够输出在屏幕上（如下图），说明读取能够进行，这两段均有读取权限。



然而，当在 start_kernel 中的添加的代码如下（有给_stext、_srodata 赋值，即尝试对.text 段和.rodata 段写入）时：

```
_stext = 1;
printk("_stext = %lx\n", _stext);
_srodata = 2;
printk("_srodata = %lx\n", _srodata);
```

屏幕上就不能输出_stext 和_srodata 的值了。如下图。说明这两段均无写入权限。



综上，.text 段有执行（X）、读取（R）权限，无写入（W）权限，所处内存页有效（V）。.rodata 段有读取（R）权限，无执行（X）、写入（W）权限，所处内存页有效（V）。这与我们对这两段的属性设置相一致，说明属性设置成功。

2. 因为在 setup_vm_final 中建立三级页表时，需要读取页表项中物理页号，转换成物理地址，再去访问下一级页表，如果不做等值映射，直接使用该物理地址将导致内存访问错误。

3. 在 setup_vm 中删去等值映射的部分，删去后代码如下：

```
void setup_vm(void) {
    /*
    1. 由于是进行 1GB 的映射 这里不需要使用多级页表
    2. 将 va 的 64bit 作为如下划分： | high bit | 9 bit | 30 bit |
        high bit 可以忽略;
```

```
6.          中间 9 bit 作为 early_pgtbl 的 index；
7.          低 30 bit 作为页内偏移。这里注意到 30 = 9 + 9 + 12，即我们只使用根页表，根页表的每个 entry 都对应 1GB 的区域。
8.      3. Page Table Entry 的权限 V | R | W | X 位设置为 1
9.      */
10.     memset(early_pgtbl,0x0,PGSIZE);
11.
12.     int index = VM_START >> 30 & 0x1ff;
13.     early_pgtbl[index] = (PHY_START >> 30 & 0x3fff) << 28 | 15;
14.
15.     return;
16. }
```

同时，将 create_mapping( )修改如下：

```
1. /**** 创建多级页表映射关系 *****/
2. /* 不要修改该接口的参数和返回值 */
3. void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, uint64 perm) {
4.     uint64 VPN[3];
5.     uint64 *page_table[3];
6.     uint64 new_page;
7.
8.     for (uint64 addr = va; addr < va + sz; addr += PGSIZE, pa += PGSIZE) {
9.         page_table[2] = pgtbl;
10.
11.        // 为每个级别的页表计算 VPN
12.        VPN[2] = (addr >> 30) & 0x1ff;
13.        VPN[1] = (addr >> 21) & 0x1ff;
14.        VPN[0] = (addr >> 12) & 0x1ff;
15.
16.        // 检查并创建每个级别的页表项
17.        for (int level = 2; level > 0; level--) {
18.            if ((page_table[level][VPN[level]] & 1) == 0) {
19.                new_page = kalloc();
20.                page_table[level][VPN[level]] = (((new_page - PA2VA_OFFSET) >> 12) << 10) | 1;
21.            }
22.            page_table[level - 1] = (uint64 *)((page_table[level][VPN[level]] >> 10) << 12 +
                                                                                    PA2VA_OFFSET);
23.        }
24.
25.        // 设置最后一级页表项
26.        page_table[0][VPN[0]] = (perm & 0b1111) | ((pa >> 12) << 10);
27.    }
28.}
```

（好像探究不出来）

五、附录

无。