

## Lab 3: RV64 虚拟内存管理

姓名：姜雨童

学号：3220103450

## 1 实验内容及原理

### 1.1 实验目的

- 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
- 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

### 1.2 实验环境

- Environment in previous labs

### 1.3 实验原理

通过两次地址映射，开启虚拟地址，通过设置页表来实现地址映射和权限控制。

虚拟内存布局为 RISC-V Linux Kernel v5.16 前 Sv39 的内存布局。

## 2 实验过程与代码实现

### 2.1 准备工作

从仓库 `src/lab3` 同步代码后，把 `lab2` 的内容加进去：

```
1 | ~/os24fall-stu$ cp -r ./src/lab2/* ./src/lab3
```

```

● jyt@fine:~/os24fall-stu$ cd src/lab3
● jyt@fine:~/os24fall-stu/src/lab3$ tree -L 8
.
├── Makefile
├── arch
│   └── riscv
│       ├── Makefile
│       ├── include
│       │   ├── clock.h
│       │   ├── defs.h
│       │   ├── mm.h
│       │   ├── printk.h
│       │   ├── proc.h
│       │   └── sbi.h
│       └── kernel
│           ├── Makefile
│           ├── clock.c
│           ├── entry.S
│           ├── head.S
│           ├── mm.c
│           ├── proc.c
│           ├── sbi.c
│           ├── trap.c
│           └── vmlinux.lds
└── fw_jump.bin
    ├── include
    │   ├── printk.h
    │   ├── stddef.h
    │   ├── stdint.h
    │   ├── stdlib.h
    │   └── string.h
    ├── init
    │   ├── Makefile
    │   ├── main.c
    │   └── test.c
    └── lib
        ├── Makefile
        ├── printk.c
        ├── rand.c
        └── string.c

```

在 `defs.h` 添加内容：

```

src > lab3 > arch > riscv > include > defs.h > ...
1  #ifndef __DEFS_H__
14
15  // lab3_add begin
16  #define OPENSBI_SIZE (0x200000)
17
18  #define VM_START (0xffffffe000000000)
19  #define VM_END (0xfffffffff0000000)
20  #define VM_SIZE (VM_END - VM_START)
21
22  #define PA2VA_OFFSET (VM_START - PHY_START)
23  // lab3_add end

```

修改 `makefile` 文件以关闭 `PIE`（位置无关执行）：在 `Makefile` 的 `CF` 中加一个 `-fno-pie`

## 2.2 开启虚拟内存映射

### 2.2.1 `setup_vm` 的实现

将从 `0x80000000` 开始的 1GB 区域进行两次映射，其中一次是等值映射（`PA == VA`），另一次是将其映射到 `direct mapping area`（使 `PA + PV2VA_OFFSET == VA`）。

为了方便访问内存，内核会预先把所有物理内存都映射至 `direct mapping area` 区域，在 RISC-V Linux Kernel 中这一段区域为 `0xffffffe000000000 ~ 0xfffffffff0000000`，因此实验中需要把 `0x80000000` 映射到 `0xffffffe000000000`。通过查询 `vmlinux.lds`/`defs.h` 可以得知对应的表示：

```

src > lab3 > arch > riscv > kernel > vmlinux.lds
1  /* 目标架构 */
2  OUTPUT_ARCH("riscv")
3
4  /* 程序入口 */
5  ENTRY(_start)
6
7  PHY_START = 0x80000000;
8  PHY_SIZE  = (128 * 1024 * 1024);
9  PHY_END   = (PHY_START + PHY_SIZE);
10 PG_SIZE   = 0x1000;
11 OPENSBI_SIZE = (0x200000);
12 VM_START    = (0xffffffe000000000);
13 VM_END      = (0xfffffffff0000000);
14 VM_SIZE     = (VM_END - VM_START);
15 PA2VA_OFFSET = (VM_START - PHY_START);
16
17 MEMORY {

```

由于本实验采用RISC-V Sv39 分页模式，因此查询[RISC-V Privileged Spec](#)后得到地址和PTE表示：

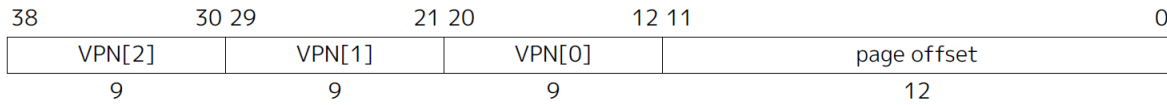


Figure 60. Sv39 virtual address.

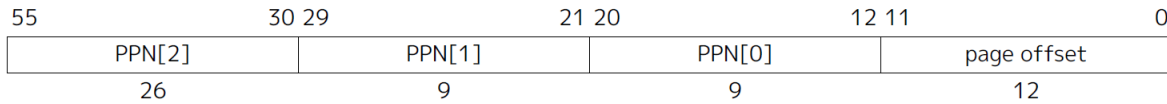


Figure 61. Sv39 physical address.

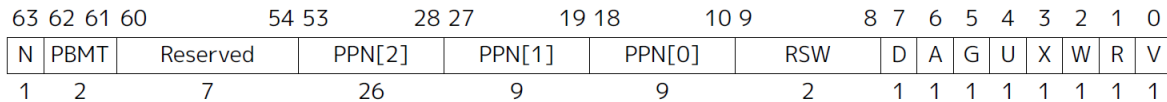


Figure 62. Sv39 page table entry.

```

1 void setup_vm() {
2     /*
3      * 1. 由于是进行 1GiB 的映射，这里不需要使用多级页表
4      * 2. 将 va 的 64bit 作为如下划分： | high bit | 9 bit | 30 bit |
5      *     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 1GiB 的区域
8      * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
9      */
10
11     unsigned long PA = PHY_START;
12     unsigned long PPN = (PA >> 12); // PPN
13     unsigned long PTE = (PPN << 10) | 0xF; // PPN, XWRV: 3-0
14
15     unsigned long VA_IDENTITY = PA;
16     int index = (VA_IDENTITY >> 30) & 0x1FF; // VPN[2]: 9 bit
17     early_pgtbl[index] = PTE;
18
19     unsigned long VA_DIRECT_MAPPING = PA + PA2VA_OFFSET;
20     index = (VA_DIRECT_MAPPING >> 30) & 0x1FF;
21     early_pgtbl[index] = PTE;
22
23     printk(GREEN "...setup_vm done.\n" CLEAR);
24 }

```

完成上述映射之后，通过 `relocate` 函数，完成对 `satp` 的设置，以及跳转到对应的虚拟地址。

实验中使用 `Sv39` 模式，因此 `MODE` 字段置八，`ASID` 字段置零，`PA >> 12 == PPN`（即 `satp` 中44位的 `PPN` 为根页表物理地址的高44位，包括了 `PPN[2]`，`PPN[1]` 和 `PPN[0]`）。`satp` 寄存器的格式如下：



Figure 56. Supervisor address translation and protection register `satp` when `SXLEN=64`, for `MODE` values Bare, Sv39, Sv48, and Sv57.

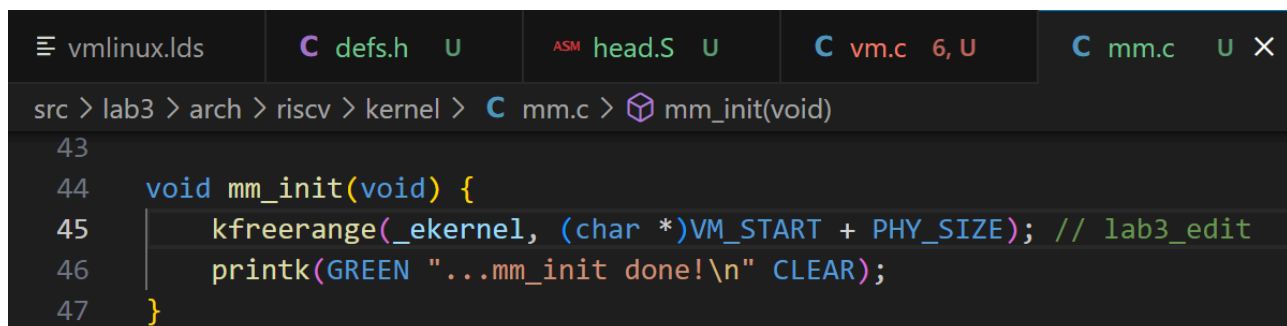
```

1  _start:
2      #(previous) initialize stack
3      la sp, boot_stack_top
4      call setup_vm // lab3_add
5      call relocate
6      call mm_init // lab2_add
7      call task_init

1  relocate:
2      # set ra = ra + PA2VA_OFFSET
3      # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
4
5      #####
6      #   YOUR CODE HERE   #
7      li t0, 0xfffffd80000000 # PA2VA_OFFSET
8      add ra, ra, t0
9      add sp, sp, t0
10     #####
11
12     # need a fence to ensure the new translations are in use
13     sfence.vma zero, zero
14
15     # set satp with early_pgtbl
16
17     #####
18     #   YOUR CODE HERE   #
19     la t1, early_pgtbl
20     srli t1, t1, 12 # PPN
21     li t2, 0x8000000000000000
22     or t1, t1, t2
23     csrw satp, t1
24     #####
25
26     ret

```

最后修改 `mm.c` 中的 `mm_init` 函数，确保结束地址为虚拟地址：



```

src > lab3 > arch > riscv > kernel > mm.c > mm_init(void)
43
44 void mm_init(void) {
45     kfreerange(_kernel, (char *)VM_START + PHY_SIZE); // lab3_edit
46     printk(GREEN "...mm_init done!\n" CLEAR);
47 }

```

本过程中需要注意的点：

- 需要给 `vm.c` 加入头文件，否则 `uint64_t` 会报错：

```

src > lab3 > arch > riscv > kernel > C vm.c > ...
1  #include "stdint.h"
2  /* early_pgtbl: 用于 setup_vm 进行 1GiB 的映射 */
3  uint64_t early_pgtbl[512] __attribute__((__aligned__(0x1000)));
4

```

- 实验中进行1GiB的映射，不需要用到多级页表，因此只需要用到VPN[2]。
- 因为物理地址和虚拟地址的大小不一样，不能简单改成虚拟地址的结束地址（下图）。

```

src > lab3 > arch > riscv > kernel > C mm.c > mm_init(void)
43
44 void mm_init(void) {
45     kfreerange(_ekernel, (char *)VM_END); // lab3_edit
46     printk(GREEN "...mm_init done!\n" CLEAR);
47 }
48

```

### 2.2.2 setup\_vm\_final 的实现

首先在 `vm.c` 中完成创建多级页表映射关系的函数 `create_mapping`：

```

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_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, uint64_t
perm) {
4      /*
5       * pgtbl 为根页表的基地址
6       * va, pa 为需要映射的虚拟地址、物理地址
7       * sz 为映射的大小，单位为字节
8       * perm 为映射的权限（即页表项的低 8 位）
9       *
10      * 创建多级页表的时候可以使用 kalloc() 来获取一页作为页表目录
11      * 可以使用 V bit 来判断页表项是否存在
12      */
13      int page_num = ceil(sz / PGSIZE);
14      for (int i = 0; i < page_num; i++)
15      {
16          uint64_t VPN[3];
17          VPN[2] = (va >> 30) & 0x1FF;
18          VPN[1] = (va >> 21) & 0x1FF;
19          VPN[0] = (va >> 12) & 0x1FF;
20          uint16_t *pt_entry = pgtbl;
21          for (int j = 2; j > 0; j--) // RWX全为0: 当前表项存储的PPN是下一层页表的物理地址
22          {
23              if ((pt_entry[VPN[j]] & 0x1) != 0) { // 使用 V bit 来判断页表项是否存在
24                  uint64_t next_PPN = pt_entry[VPN[j]] >> 10 & 0xFFFFFFFF; // 44 bit
25                  pt_entry = (uint16_t *)((next_PPN << 12) + PA2VA_OFFSET);
26              } else {
27                  uint64_t new_pt_entry = kalloc(); // 使用 kalloc来获取一页作为页表目录

```

```

28         uint64_t next_PPN = ((new_pt_entry - PA2VA_OFFSET) >> 12) &
0xFFFFFFFFFFFF;
29         pt_entry[VPN[j]] = next_PPN << 10 | 0x1; // 设置 V bit
30         pt_entry = (uint64_t *)new_pt_entry;
31     }
32 }
33 pt_entry[VPN[0]] = ((pa >> 12) & 0xFFFFFFFFFFFF) << 10 | perm;
34 va += PGSIZE;
35 pa += PGSIZE;
36 }
37 }

```

过程中需要注意的点：

- 页表结构使用的是虚拟地址，因此在设置表项中 `PPN` 时要减去偏移量。
- 多级页表内部映射时，通过表项内 `XWR` 三位均为零来表示当前表项存储的 PPN 是下一层页表的物理地址，因此和 `perm` 参数无关；最后映射到物理地址时，使用 `perm` 来设置映射权限（即页表项的低 8 位）。
- `&` 的运算优先级比 `<<`/`>>` 高，因此要注意使用括号使运算顺序符合要求。

接下来在 `setup_vm_final` 中完成对所有物理内存 (128M) 的映射：

```

1  /* swapper_pg_dir: kernel pagetable 根目录, 在 setup_vm_final 进行映射 */
2  uint64_t swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
3  extern char _stext[], _etext[];
4  extern char _srodata[], _erodata[];
5  extern char _sdata[];
6  void setup_vm_final() {
7      memset(swapper_pg_dir, 0x0, PGSIZE);
8
9      // No OpenSBI mapping required
10
11      // mapping kernel text X|-|R|V
12      create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_stext, (uint64_t)_stext -
PA2VA_OFFSET, (uint64_t)(_etext - _stext), 0xb);
13
14      // mapping kernel rodata -|-|R|V
15      create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_srodata, (uint64_t)_srodata -
PA2VA_OFFSET, (uint64_t)(_erodata - _srodata), 0x3);
16
17      // mapping other memory -|W|R|V
18      create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_sdata, (uint64_t)_sdata -
PA2VA_OFFSET, (PHY_END + PA2VA_OFFSET - (uint64_t)_sdata), 0x7);
19
20      // set satp with swapper_pg_dir
21      unsigned long new_satp = 0x8000000000000000 | (((unsigned long)swapper_pg_dir -
PA2VA_OFFSET) >> 12);
22      csr_write(satp, new_satp);
23
24      // YOUR CODE HERE
25
26      // flush TLB
27      asm volatile("sfence.vma zero, zero");
28      return;
29 }

```

并在 `head.S` 中调用 `setup_vm_final`：由于 `setup_vm_final` 函数中调用了 `kalloc` 函数来获取新的页，在调用前应该完成内存管理的初始化。

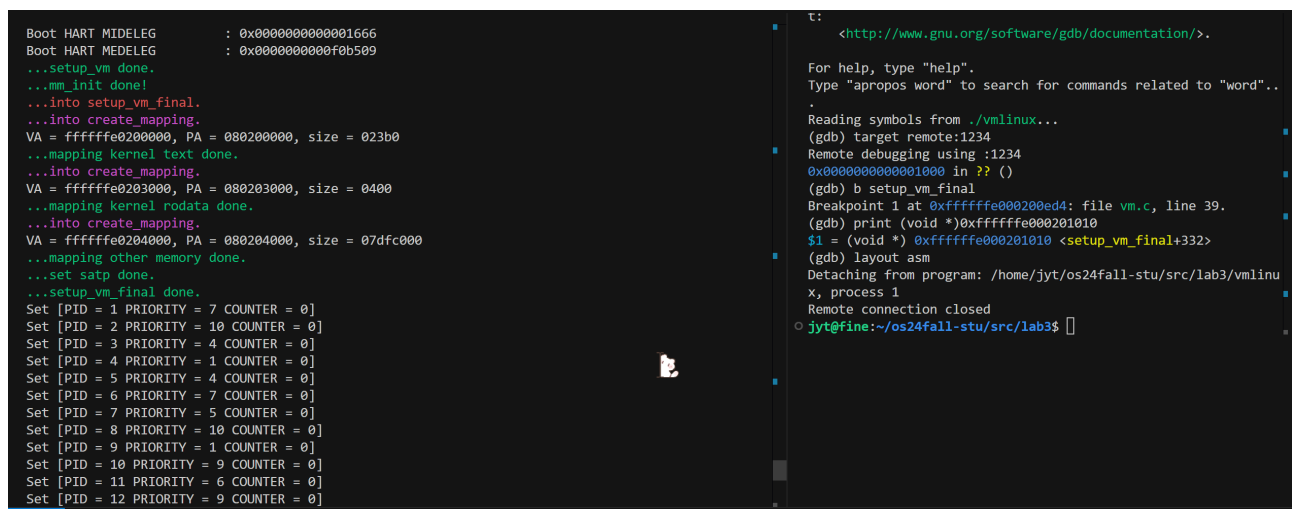
```

1      #(previous) initialize stack
2      la sp, boot_stack_top
3
4      call setup_vm // lab3_add
5      call relocate
6      call mm_init // lab2_add
7      call setup_vm_final
8      call task_init

```

### 3 编译与测试

编译通过，输出结果符合预期。



```

Boot HART MIDELEG      : 0x0000000000001666
Boot HART MEDELEG      : 0x0000000000f0b509
...setup_vm done.
...mm_init done!
...into setup_vm_final.
...into create_mapping.
VA = fffffffe0200000, PA = 080200000, size = 023b0
...mapping kernel text done.
...into create_mapping.
VA = fffffffe0203000, PA = 080203000, size = 0400
...mapping kernel rodata done.
...into create_mapping.
VA = fffffffe0204000, PA = 080204000, size = 07dfc000
...mapping other memory done.
...set satp done.
...setup_vm_final done.
Set [PID = 1 PRIORITY = 7 COUNTER = 0]
Set [PID = 2 PRIORITY = 10 COUNTER = 0]
Set [PID = 3 PRIORITY = 4 COUNTER = 0]
Set [PID = 4 PRIORITY = 1 COUNTER = 0]
Set [PID = 5 PRIORITY = 4 COUNTER = 0]
Set [PID = 6 PRIORITY = 7 COUNTER = 0]
Set [PID = 7 PRIORITY = 5 COUNTER = 0]
Set [PID = 8 PRIORITY = 10 COUNTER = 0]
Set [PID = 9 PRIORITY = 1 COUNTER = 0]
Set [PID = 10 PRIORITY = 9 COUNTER = 0]
Set [PID = 11 PRIORITY = 6 COUNTER = 0]
Set [PID = 12 PRIORITY = 9 COUNTER = 0]

```

```

t:  <http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"..
.
Reading symbols from ./vmlinux...
(gdb) target remote:1234
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb) b setup_vm_final
Breakpoint 1 at 0xffffffe000200ed4: file vm.c, line 39.
(gdb) print (void *)0xffffffe000201010
$1 = (void *) 0xffffffe000201010 <setup_vm_final+332>
(gdb) layout asm
Detaching from program: /home/jyt/os24fall-stu/src/lab3/vmlinu
X, process 1
Remote connection closed
jyt@fine:~/os24fall-stu/src/lab3$

```

### 4 实验中遇到的问题及解决方法

其他的小问题在上面写过，这里不做赘述，只记录一个困扰了我很久的问题：编译测试的时候发现一直在重复初始化，更进一步打印出具体信息后发现执行到 `setup_vm_final` 中 `mapping` 部分后就会跳转到 `setup_vm` 重新执行并陷入死循环（图4-1），并不执行设置 `satp` 的部分或是在此处报错（图4-2）。



```

Boot HART MIDELEG      : 0x0000000000001666
Boot HART MEDELEG      : 0x0000000000f0b509

...setup_vm done.
...mm_init done!
...into setup_vm_final.
...into create_mapping.
VA = fffffffe0200000, PA = 080200000, size = 023a4
...mapping kernel text done.
...into create_mapping.
VA = fffffffe0203000, PA = 080203000, size = 0400
...mapping kernel rodata done.
...into create_mapping.
VA = fffffffe0204000, PA = 080204000, size = 07dfc000
...mapping other memory done.

...setup_vm done.
...mm_init done!
...into setup_vm_final.
...into create_mapping.
VA = fffffffe0200000, PA = 080200000, size = 023a4
...mapping kernel text done.
...into create_mapping.
VA = fffffffe0203000, PA = 080203000, size = 0400
...mapping kernel rodata done.
...into create_mapping.
VA = fffffffe0204000, PA = 080204000, size = 07dfc000
...mapping other memory done.

...setup_vm done.

```

图4-1

```

// mapping kernel text X|-|R|V
create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_stext, (uint64_t)_stext - PA2VA_OFFSET, (uint64_t)(_etext -
printf(GREEN "...mapping kernel text done.\n" CLEAR); // test*
// mapping kernel rodata -|-|R|V
create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_srodata, (uint64_t)_srodata - PA2VA_OFFSET, (uint64_t)(_erod
printf(GREEN "...mapping kernel rodata done.\n" CLEAR); // test*
// mapping other memory -|W|R|V
create_mapping((uint64_t *)swapper_pg_dir, (uint64_t)_sdata, (uint64_t)_sdata - PA2VA_OFFSET, (PHY_END + PA2VA_OF
printf(GREEN "...mapping other memory done.\n" CLEAR); // test*

// set satp with swapper_pg_dir
// unsigned long new_satp = 0x8000000000000000 | (((unsigned long)swapper_pg_dir - PA2VA_OFFSET) >> 12);
unsigned long new_satp = (0x1 << 63) | (((unsigned long)swapper_pg_dir - PA2VA_OFFSET) >> 12); // WRONG!!!
csr_write(satp, new_satp);
printf(GREEN "...set satp done.\n" CLEAR); // test*

```

图4-2

使用 `gdb` debug 的时候发现执行到设置 `satp` 的指令时，再执行一步会报错 `Cannot access memory at address ...`，且无法到达的地址正是下一条指令所在的地址（图4-3，图中 `csrwr` 指令的地址是 `0x...20100C`，下一条 `auipc` 指令的地址是 `0x...201010`，而执行过 `csrwr` 指令后报错正是无法到达 `0x...201010`）。

```

Register group: general
zero      0x0      0
ra         0xffffffff000200fe4      0xffffffff000200fe4 <setup_vm_final+2
sp         0xffffffff000205fe0      0xffffffff000205fe0
gp         0x0      0x0
tp         0x80047000      0x80047000
t0         0xffffffffd80000000      -139586437120
t1         0x0      0
t2         0x0      0
fp         0xffffffff000206000      0xffffffff000206000 <kmem>
s1         0x1      1
a0         0x27     39

0xffffffff000200ff4 <setup_vm_final+304> add    a5,a4,a5
0xffffffff000200ff8 <setup_vm_final+308> srli   a5,a5,0xc
0xffffffff000200ffc <setup_vm_final+312> sd     a5,-24(s0)
0xffffffff000201000 <setup_vm_final+316> ld     a5,-24(s0)
0xffffffff000201004 <setup_vm_final+320> sd     a5,-32(s0)
0xffffffff000201008 <setup_vm_final+324> ld     a5,-32(s0)
0xffffffff00020100c <setup_vm_final+328> csrw   satp,a5
0xffffffff000201010 <setup_vm_final+332> auipc  a0,0x2
0xffffffff000201014 <setup_vm_final+336> addi   a0,a0,720
0xffffffff000201018 <setup_vm_final+340> jal    0xffffffff000202214 <printk>

```

```

Register group: general
zero      0x0      0
ra         0xffffffff000200fe4      0xffffffff000200fe4 <setup_vm_final+2
sp         0xffffffff000205fe0      0xffffffff000205fe0
gp         0x0      0x0
tp         0x80047000      0x80047000
t0         0xffffffffd80000000      -139586437120
t1         0x0      0
t2         0x0      0
fp         0xffffffff000206000      0xffffffff000206000 <kmem>
s1         0x1      1
a0         0x27     39

```

```

remote Thread 1.1 (asm) In: setup_vm_final      L55  PC: 0xffffffff000201008
Run till exit from #0  create_mapping (
  pgtbl=0xffffffff000208000 <swapper_pg_dir>, va=18446743936272711680,
  pa=2149597184, sz=132104192, perm=7) at vm.c:80
setup_vm_final () at vm.c:52
(gdb) si
printk (s=0xffffffff000208000 <swapper_pg_dir> "") at printk.c:285
(gdb) finish
Run till exit from #0  printk (s=0xffffffff000208000 <swapper_pg_dir> "")
  at printk.c:285
setup_vm_final () at vm.c:54
Value returned is $5 = 39
(gdb) si
(gdb) .

```

```

remote Thread 1.1 (asm) In: setup_vm_final      L56  PC: 0xffffffff000201010
pgtbl=0xffffffff000208000 <swapper_pg_dir>, va=18446743936272711680,
pa=2149597184, sz=132104192, perm=7) at vm.c:80
setup_vm_final () at vm.c:52
(gdb) si
printk (s=0xffffffff000208000 <swapper_pg_dir> "") at printk.c:285
(gdb) finish
Run till exit from #0  printk (s=0xffffffff000208000 <swapper_pg_dir> "")
  at printk.c:285
setup_vm_final () at vm.c:54
Value returned is $5 = 39
(gdb) si
Cannot access memory at address 0xffffffff000201010
(gdb) .

```

图4-3

继续执行会再次报错，然后跳转到程序开头重新执行，也就导致了编译测试的时候出现死循环的情况（图4-4）。

```
root
v1.12
rv64imafdc
time,sstc
16
4
54
16
0x0000000000001666
0x0000000000f0b509

0200000, pa = 08020000
e.
0203000, pa = 08020300
one.
0204000, pa = 08020400
ne.

remote Thread 1.1 (asm) In: setup_vm_final L39 PC: 0xffffffff000200ed4
(gdb) n
(gdb) si
(gdb) n
(gdb) si
(gdb) n
(gdb) si
(gdb) si
Cannot access memory at address 0xffffffff000201010
Cannot access memory at address 0xffffffff000201010
(gdb) c
Continuing.

Breakpoint 2, setup_vm_final () at vm.c:39
(gdb) 
```

图4-4

最后发现是设置 `satp` 指令的值时出错（虽然还并没有理解为什么会出现这种错误）：

```
// set satp with swapper pg dir
unsigned long new_satp = 0x8000000000000000 | (((unsigned long)swapper_pg_dir - PA2VA_OFFSET) >> 12);
unsigned long new_satp = (0x1 << 63) | (((unsigned long)swapper_pg_dir - PA2VA_OFFSET) >> 12);
csr_write(satp, new_satp);
printk(GREEN "...set satp done.\n" CLEAR); // test*

// YOUR CODE HERE
```

## 为什么第一行不会报错而第二行会报错？

理论上，从表达式的角度来看，这两行的结果是相同的。然而，实际报错的原因可能与以下几个因素有关：

### 1. 编译器优化或设置：

- 如果在某些情况下，编译器无法识别 `(0x1 << 63)` 并优化它，那么它会导致溢出或未定义行为。这通常发生在某些特定的编译器设置或架构上。

### 2. 数据类型：

- 如果 `unsigned long` 在特定平台上为 32 位，左移 63 位的结果将会导致溢出，因为 `1 << 63` 已经超出了 `unsigned long` 的最大位数。在这个情况下，`0x1 << 63` 会导致未定义行为或者编译错误，而 `0x8000000000000000` 本身是一个合法的值。

### 3. 语法或其他上下文错误：

- 如果在某些环境下，特定的语法错误、上下文问题或使用了不适当的宏、类型定义等，可能会导致一行代码有效而另一行无效。

## 5 思考题

### 5.1 验证 `.text`，`.rodata` 段的属性是否成功设置，给出截图。

在 `main.c/start_kernel` 中加入代码段输出这两个属性的值，发现能够成功输出：

```
1 printk(YELLOW "_stext = %x\n" CLEAR, _stext);
2 printk(YELLOW "_srodata = %x\n" CLEAR, _srodata);
```

```
Set [PID = 30 PRIORITY = 10 COUNTER = 0]
Set [PID = 31 PRIORITY = 3 COUNTER = 0]
...task_init done!
2024 ZJU Operating System
_stext = 6117
_srodata = 32335b1b
Set [PID = 31 PRIORITY = 3 COUNTER = 3]
Set [PID = 30 PRIORITY = 10 COUNTER = 10]
```

因为成功设置后，这两个属性都不允许写入值，因此加入修改的代码段再次输出，发现程序在这里会先卡住，并不修改值也不会再次输出，说明 `_stext` 成功设置了：

```
printk(YELLOW "_stext = %x\n" CLEAR, _stext);
printk(YELLOW "_srodata = %x\n" CLEAR, _srodata);

_stext = 0xFF;
printk(YELLOW "_stext = %x\n" CLEAR, _stext);

...task_init done!
2024 ZJU Operating System
_stext = 6117
_srodata = 32335b1b
QEMU: Terminated
```

如果在 `trap.c/trap_handler` 中对异常打印出错误信息，会出现报错：

```
32         break;
33     }
34 } else { // exception
35     printk(RED "[Exception]\n" CLEAR); // [TODO]
36 }
```

问题 输出 调试控制台 终端 端口 1

```
[Exception]
[Exception]
[Exception]
[Exception]
[Exception]
[Exception]
[Exception]
[Exception]
```

修改 `_srodata` 也是同样的结果：

```
printk(YELLOW "_stext = %x\n" CLEAR, _stext);
printk(YELLOW "_srodata = %x\n" CLEAR, _srodata);

// _stext = 0xFF;
// printk(YELLOW "_stext = %x\n" CLEAR, _stext);

_srodata = 0xFF;
printk(YELLOW "_srodata = %x\n" CLEAR, _srodata);
```

问题 输出 调试控制台 终端 端口 1

```
[Exception]
[Exception]
[Exception]
[Exception]
[Exception]
[Exception]
[Exception]
```

5.2 为什么我们在 `setup_vm` 中需要做等值映射？在 **Linux** 中，是不需要做等值映射的，请探索一下不在 `setup_vm` 中做等值映射的方法。

- 本次实验中如果不做等值映射，会出现什么问题，原因是什么？

会导致内存访问错误。我们在 `relocate` 中设置了 `satp` 开启虚拟地址，但并没有改变 `pc` 的值，于是该物理地址会被误认为是虚拟地址。如果没有等值映射，就无法正确访问对应地址，会出现内存访问错误导致无法执行 `ret` 退出 `relocate` 函数。而在 `setup_vm_final` 中建立三级页表时，从 `PTE` 得到的 `PPN` 也

- 简要分析 [Linux v5.2.21](#) 或之后的版本中的内核启动部分（直至 `init/main.c` 中 `start_kernel` 开始之前），特别是设置 `satp` 切换页表附近的逻辑；

```

56      /* Initialize page tables and relocate to virtual addresses */
57      la sp, init_thread_union + THREAD_SIZE
58      call setup_vm      初始化页表，设置虚拟内存并进行重定位
59      call relocate
60
61      /* Restore C environment */
62      la tp, init_task
63      sw zero, TASK_TI_CPU(tp)
64      la sp, init_thread_union + THREAD_SIZE
65
66      /* Start the kernel */
67      mv a0, s1
68      call parse_dtb      start_kernel启动内核
69      tail start_kernel

```

```

71      relocate:
72      /* Relocate return address */
73      li a1, PAGE_OFFSET
74      la a0, _start      重定位返回地址ra
75      sub a1, a1, a0      a1是VA2PA_OFFSET
76      add ra, ra, a1
77
78      /* Point stvec to virtual address of instruction after satp write */
79      la a0, 1f          加载 (forward) 标签1的指令地址
80      add a0, a0, a1      设置中断向量，把设置satp后的
81      csrwr CSR_STVEC, a0 指令的虚拟地址存入STVEC
82
83      /* Compute satp for kernel page tables, but don't load it yet */
84      la a2, swapper_pg_dir
85      srl a2, a2, PAGE_SHIFT  计算用于内核的satp，暂存于a2中
86      li a1, SATP_MODE
87      or a2, a2, a1
88
89      /*
90       * Load trampoline page directory, which will cause us to trap to
91       * stvec if VA != PA, or simply fall through if VA == PA. We need a
92       * full fence here because setup_vm() just wrote these PTEs and we n
93       * to ensure the new translations are in use.
94       trampoline_pg_dir 是一个临时页表目录，用来处理从内核到用户空间的切换
95       la a0, trampoline_pg_dir 计算satp并存入寄存器
96       srl a0, a0, PAGE_SHIFT    即当前satp中信息与临时页表有关
97       or a0, a0, a1
98       sfence.vma 确保之前写入的页表项得到同步与生效 VA!=PA时，trap，进入
99       csrwr CSR_SATP, a0      stvec存的trap handler地址，即1:
100
101      .align 2
102      1:
103      /* Set trap vector to spin forever to help debug */
104      la a0, .Lsecondary_park
105      csrwr CSR_STVEC, a0      设置trap handler地址到一个无限循环地址，
                               便于debug

```

```

106      /* Reload the global pointer */
107      .option push
108      .option norelax
109      la gp, __global_pointer$
110      .option pop
111
112      /*
113       * Switch to kernel page tables. A full fence is necessary in order to
114       * avoid using the trampoline translations, which are only correct for
115       * the first superpage. Fetching the fence is guaranteed to work
116       * because that first superpage is translated the same way.
117       */
118      csr CSR_SATP, a2 把计算过的satp存入寄存器，使程序能切换到
119      sfence.vma        内核页表
120                        确保之前写入的页表项得到同步与生效
121      ret
122

```

- 回答 Linux 为什么可以不进行等值映射，它是在无等值映射的情况下让 pc 从物理地址跳到虚拟地址；

Linux使用了中断来解决无等值映射的问题。pc的物理地址没有等值映射到虚拟地址上（`VA!=PA`），访问地址会出现异常，使程序陷入中断，跳转到 `stvec` 中保存的 `trap_handler` 地址（即设置 `stap` 的后续指令）。

- Linux v5.2.21 中的 `trampoline_pg_dir` 和 `swapper_pg_dir` 有什么区别，它们分别是在哪里通过 `satp` 设为所使用的页表的；

前者是一个临时页表目录，用来处理从内核到用户空间的切换，确保程序能正确访问内存，可以对标本次实验中的 `early_pgtbl`。它在代码第99行（设置 `satp` 为内核页表之前，相当于通过这一过程将程序从使用物理地址切换到使用虚拟地址）通过 `satp` 设为所使用的页表。

后者是内核使用的页表目录，用来管理内核的虚拟地址，确保内核初始化之后物理地址和虚拟地址能够正确转换，可以对标本次实验中的 `swapper_pg_dir`。它在代码第118行（内核初始化后期）通过 `satp` 设为所使用的页表。

- 尝试修改你的 kernel，使得其可以像 Linux 一样不需要等值映射。

将等值映射注释化，并在 `head.S` 中加入中断处理，程序能正常运行：

```

1      # trap
2      la t3, 1f
3      add t3, t3, t0 # 转化成虚拟地址
4      csr stvec, t3
5
6      la t1, early_pgtbl
7      srli t1, t1, 12 # PPN
8      li t2, 0x8000000000000000
9      or t1, t1, t2
10     csr satp, t1
11
12 1:
13     la t4, _traps
14     csr stvec, t4
15     sfence.vma zero, zero

```



```

...setup_vm done.
...mm_init done!
...into setup_vm_final.
...into create_mapping.
VA = fffffffe0200000, PA = 080200000, size = 02380
...mapping kernel text done.
...into create_mapping.
VA = fffffffe0203000, PA = 080203000, size = 0400
...mapping kernel rodata done.
...into create_mapping.
VA = fffffffe0204000, PA = 080204000, size = 07dfc000
...mapping other memory done.
...set satp done.
...setup_vm_final done.
...task_init done!
2024 ZJU Operating System
_stext = 6117
_srodata = 32335b1b
QEMU: Terminated
jyt@fine:~/os24fall-stu/src/lab3$

Reading symbols from ./vmlinuz...
(gdb) target remote:1234
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb) b setup_vm_final
Breakpoint 1 at 0xffffffe000200ed4:
9.

17     int index;
18     unsigned long PA = PHY_START;
19     unsigned long PPN = (PA >> 12); // PPN
20     unsigned long PTE = (PPN << 10) | 0xF; // PPN, XWRV: 3-0
21
22     // unsigned long VA_IDENTITY = PA;
23     // index = (VA_IDENTITY >> 30) & 0x1FF; // VPN[2]: 9 bit
24     // early_pgtbl[index] = PTE;
25
26     unsigned long VA_DIRECT_MAPPING = PA + PA2VA_OFFSET;
27     index = (VA_DIRECT_MAPPING >> 30) & 0x1FF;
28     early_pgtbl[index] = PTE;

```

## 6 心得体会

实验本身对我而言有点难度，理解起来有点绕，但是一步一步也确实加深了我对虚拟内存的理解。就是半路杀出来一个bug，花了相当久的时间才找出来，整个实验又更让人头大了一点。除此之外，我认为实验中最难的部分在于物理地址和虚拟地址的转换，因为是手动转换的，所以编程时要格外注意需要的是物理地址还是虚拟地址，是否需要加/减偏移量等。如果出错会导致程序跑到错误的地址，无法执行下去，这时通过gdb来追踪执行指令的地址可以找出问题。