

浙江大学

操作系统实验三

课程名称:	操作系统
作业名称:	RV64 虚拟内存管理
姓 名:	仇国智
学 号:	3220102181
电子邮箱:	3220102181@zju.edu.cn
联系电话:	13714176104
指导教师:	申文博

2024 年 11 月 4 日

目录

1	实验内容及原理	1
1.1	实验内容	1
1.2	实验原理	1
2	实验具体过程与代码实现	4
2.1	准备工程	4
2.2	setup_vm 的实现	5
2.3	setup_vm_final 的实现	9
3	实验结果与分析	15
4	遇到的问题及解决方法	17
5	总结与心得	17
6	思考题	17
6.1	验证.text,.rodata 段的属性是否成功设置, 给出截图.	17
6.2	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. 你需要回答以下问题:	21

1 实验内容及原理

1.1 实验内容

- 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 实验原理

前言

在 Lab2 中我们赋予了操作系统对多个线程调度以及并发执行的能力, 由于目前这些线程都是内核线程, 因此他们可以共享运行空间, 即运行不同线程对空间的修改是相互可见的. 但是如果我们需要线程相互隔离, 以及在多线程的情况下更加高效的使用内存, 就必须引入虚拟内存这个概念.

虚拟内存可以为正在运行的进程提供独立的内存空间, 制造一种每个进程的内存都是独立的假象. 同时虚拟内存到物理内存的映射也包含了对内存的访问权限, 方便内核完成权限检查.

在本次实验中, 我们需要关注内核如何开启虚拟地址以及通过设置页表来实现地址映射和权限控制.

satp 寄存器

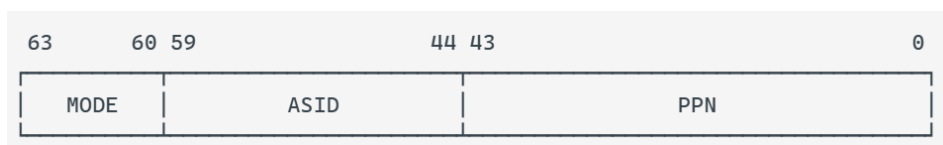


图 1: satp 寄存器

- MODE 字段:

SXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section 10.3).
SXLEN=64		
Value	Name	Description
0	Bare	No translation or protection.
1-7	-	<i>Reserved for standard use</i>
8	Sv39	Page-based 39-bit virtual addressing (see Section 10.4).
9	Sv48	Page-based 48-bit virtual addressing (see Section 10.5).
10	Sv57	Page-based 57-bit virtual addressing (see Section 10.6).
11	Sv64	<i>Reserved for page-based 64-bit virtual addressing.</i>
12-13	-	<i>Reserved for standard use</i>
14-15	-	<i>Designated for custom use</i>

图 2: MODE 字段

- ASID 字段: 本实验中不使用
- PPN 字段: 页表的物理页号

Sv39 分页模式

Sv39 模式定义物理地址有 56 位, 虚拟地址有 64 位. 虚拟地址的 64 位只有低 39 位有效, 其 63-39 位为 0 时代表 user space address, 为 1 时代表 kernel space address.

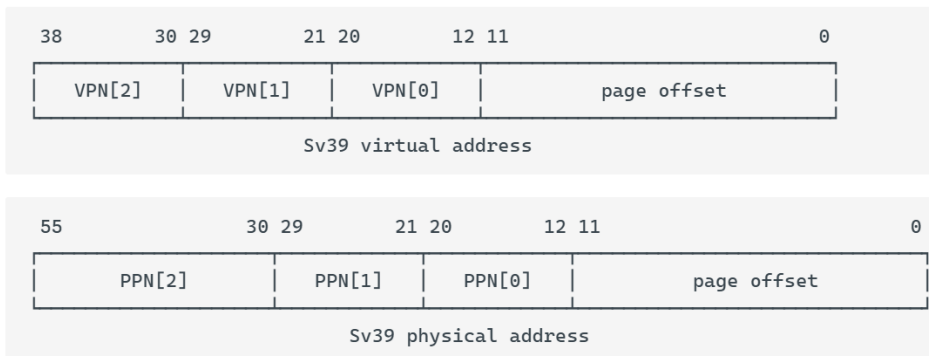


图 3: Sv39 分页模式的物理地址和虚拟地址

Sv39 支持三级页表结构, VPN[2] VPN[1] VPN[0] (Virtual Page Number) 分别代表每级页表的虚拟页号, PPN[2] PPN[1] PPN[0] (Physical Page Number) 分别代表每级页表的物理页号, 物理地址和虚拟地址的低 12 位表示页内偏移 (page offset).

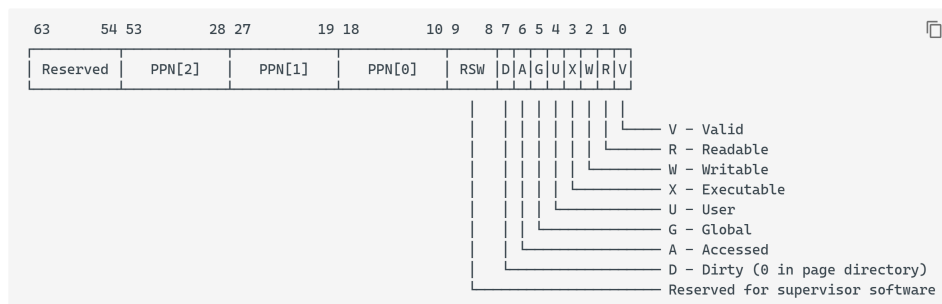


图 4: Sv39 分页模式的页表项

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	<i>Reserved for future use.</i>
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	<i>Reserved for future use.</i>
1	1	1	Read-write-execute page.

图 5: Sv39 分页模式的 RWX 字段编码规则

Sv39 虚拟地址转换

虚拟地址转化为物理地址流程图如下:

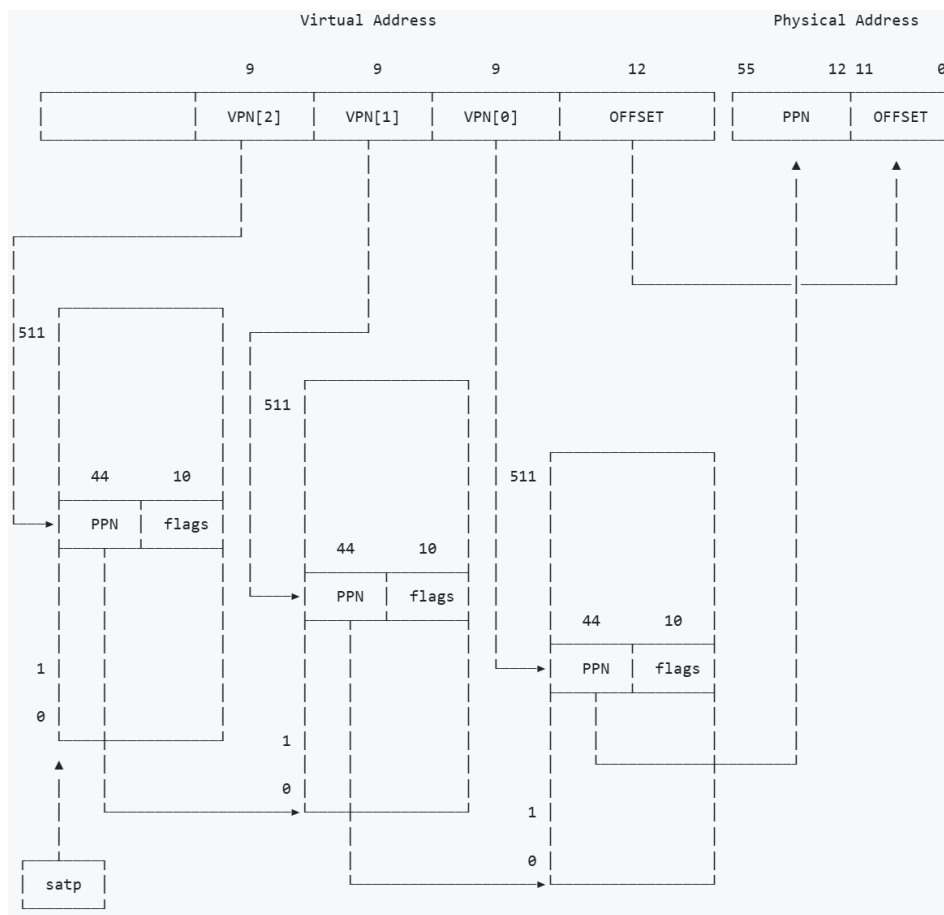


图 6: Sv39 虚拟地址转换

2 实验具体过程与代码实现

2.1 准备工程

修改 defs.h, 在 defs.h 添加如下内容:

```

1  #define OPENSBI_SIZE (0x200000)
2
3  #define VM_START (0xffffffe000000000)
4  #define VM_END (0xfffffffff00000000)
5  #define VM_SIZE (VM_END - VM_START)
6
7  #define PA2VA_OFFSET (VM_START - PHY_START)

```

在项目顶层目录的 Makefile 中需要做如下更改, 使用刷新缓存的指令扩展:

```

1    ...
2    ISA := rv64imafd_zifencei
3    ...

```

在 Makefile 中加上 -fno-pie 选项, 禁用位置无关代码, 防止地址被设置为虚拟地址后干扰启动过程.

之后启动内核, 正常运行, 通过 lab2 的测试.

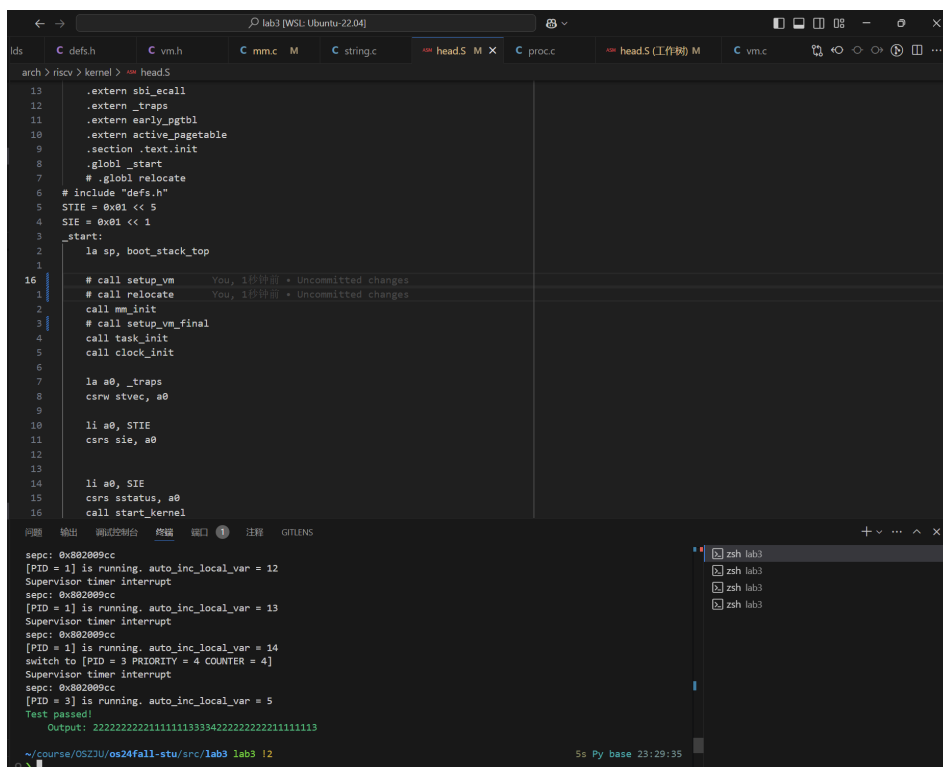


图 7: 准备工程阶段 lab2 测试

2.2 setup_vm 的实现

首先我们在 vm.h 文件中实现了一些宏定义, 用于方便我们进行页表的操作, 主要涉及 VA, PA, VPN, PPN, PTE, flag 之间的转换.

```

1    #ifndef __VM_H__
2    #define __VM_H__
3
4    #include "stdint.h"
5    #include "defs.h"
6    #define VPN(va, level) ((va) >> (39 - (level)*9) & 0x1ff)

```

```

7 // sizelevel: 1 for 1GB, 2 for 2MB, 3 for 4KB
8 // #define MAKEPTE(ppn,flag,sizelevel) (((ppn) <<
    37-9*(sizelevel) | (flag))
9 #define ADDR2PPN(addr,sizelevel) (((addr) >>
    (39-9*(sizelevel)))&(0xffffffffffff>>(47-9*(sizelevel))))
10 #define PPN2ADDR(ppn,sizelevel) (((ppn) << (39-9*(sizelevel))))
11 #define PTE2PPN(pte,sizelevel) (((pte) >>
    (37-9*(sizelevel)))&(0xffffffffffff>>(47-9*(sizelevel))))
12 #define PTE2FLAG(pte) ((pte) & 0x1ff)
13 #define PTE2ADDR(pte,sizelevel)
    (PPN2ADDR(PTE2PPN(pte,sizelevel),sizelevel))
14 #define PPN2PTE(ppn,flag,sizelevel) (((ppn) <<
    (37-9*(sizelevel))) | (flag))
15 #define ADDR2PTE(addr,flag,sizelevel)
    (PPN2PTE(ADDR2PPN(addr,sizelevel),flag,sizelevel))
16 #define PA2VA(addr) ((addr) - PHY_START + VM_START)
17 #define VA2PA(addr) ((addr) - VM_START + PHY_START)
18 typedef uint64_t pte_t;
19 typedef pte_t (*pagetable_ptr_t)[512];
20
21 #define PTE_V 0x001
22 #define PTE_R 0x002
23 #define PTE_W 0x004
24 #define PTE_X 0x008
25 #define PTE_U 0x010
26 #define PTE_G 0x020
27 #define PTE_A 0x040
28 #define PTE_D 0x080
29
30 #endif

```

然后我们完成 `setup_vm` 函数, 该函数的作用配置页表, 并且在页表中将物理地址映射到和物理地址相同的虚拟地址和我们预计内核要转移到的虚拟地址. 并且我们使用了 1G 的页, 只需第一级页表就可以完成映射, 这样我们就不用涉及多级页表的内存页申请和释放问题 (此时负责管理空闲页的 `mm` 的模块还没有初始化, 因为空闲页是

使用链表管理的, 使用虚拟地址后再初始化便于之后使用).

```
1  void setup_vm(void)
2  {
3      memset(early_pgtbl, 0, PGSIZE);
4      pte_t *pte;
5      int level = 1;
6      pte = (pte_t *)&early_pgtbl[VPN(VM_START, level)];
7      *pte = ADDR2PTE(PHY_START, PTE_V | PTE_R | PTE_W | PTE_X, 1);
8      pte = (pte_t *)&early_pgtbl[VPN(PHY_START, level)];
9      *pte = ADDR2PTE(PHY_START, PTE_V | PTE_R | PTE_W | PTE_X, 1);
10     // printk("%d",1);
11 }
```

之后就是要进行页表项的切换, 实现 `relocate` 函数. 因为在切换过程中涉及 `sp`, `ra` 等寄存器的变化, 所以我们直接使用汇编代码进行切换, 注意在这个函数返回完成后内核就会进入预计的虚拟地址. 在切换前后, 同样的虚拟地址对应的物理地址可能会发生变化, 所以我们需要在切换前后进行 `cache` 的刷新.

```
1  relocate:
2  # set ra = ra + PA2VA_OFFSET
3  # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
4
5  li t1, PA2VA_OFFSET
6  add ra, ra, t1
7  add sp, sp, t1
8
9  # set satp with early_pgtbl
10
11  la t0, early_pgtbl # PPN
12  srli t1, t0, 12
13  li t2, 8 # MODE
14  slli t2, t2, 60
15  or t1, t1, t2
16  li t2, 0xf0000fffffffffff # ASID
17  and t1, t1, t2
18
```

```
19    # la t3, active_pagetable
20    # sd t0, 0(t3)
21
22    # flush d.cache
23    fence
24
25    # flush tlb
26    sfence.vma zero, zero
27
28    # flush icache
29    fence.i
30
31    csrw satp, t1
32
33    # flush d.cache
34    fence
35
36    # flush tlb
37    sfence.vma zero, zero
38
39    # flush icache
40    fence.i
41
42    ret
```

最后启动内核, 正常运行, 通过 lab2 的测试.

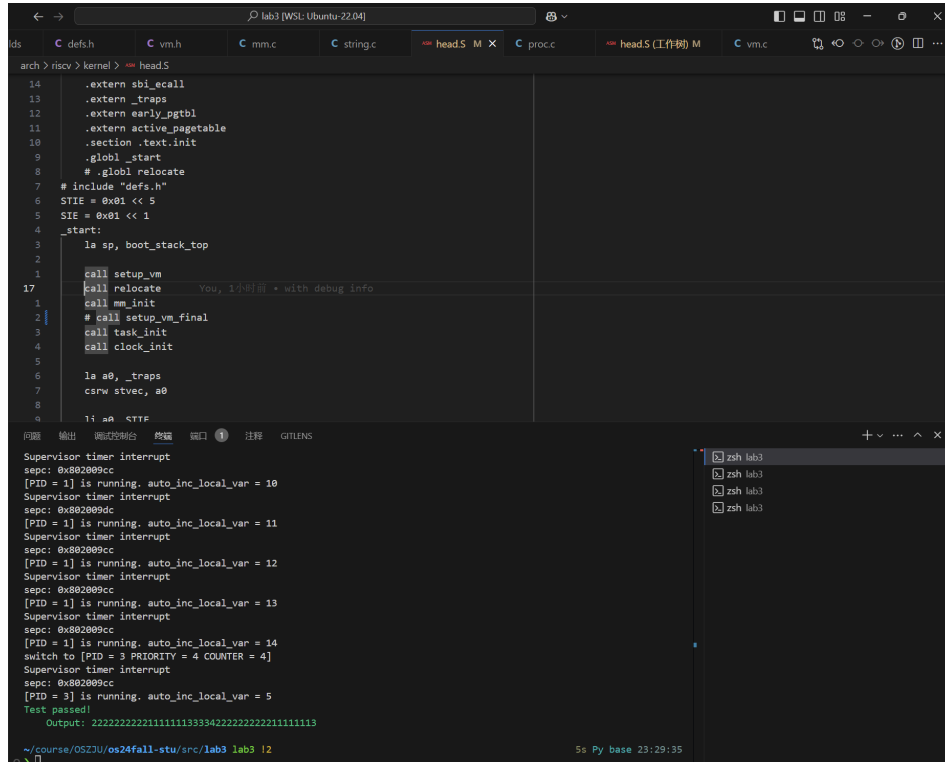


图 8: setup_vm 阶段 lab2 测试

2.3 setup_vm_final 的实现

我们在完成上面的程序后，内核已经进入了预计的虚拟地址。但是此时地址空间的管理是很粗粒度的，我们需要进一步的细化，需要采用 4k 的页，并且需要对不同的段进行权限设置。由于我上过 MIT6.S081 的课，所以我采用 walk, setup_vm_final, create_mapping, set_pgtbl 来实现这个功能。

我们在 setup_vm_final 中配置每个段的权限和范围，并调用 create_mapping 来完成页表的具体配置。

```

1  void setup_vm_final(void)
2  {
3      memset((uint64_t *)swapper_pg_dir, 0x0, PGSIZE);
4
5      // No OpenSBI mapping required
6
7      // mapping kernel text X|-|R|V
8      create_mapping((uint64_t *)swapper_pg_dir, _stext,
                     VA2PA(_stext), PGROUNDUP(_etext - _stext), PTE_R | PTE_X |

```

```

PTE_V);
9
10 // mapping kernel rodata -|-|R|V
11 create_mapping((uint64_t *)swapper_pg_dir, _srodata,
    VA2PA(_srodata), PGROUNDUP(_erodata - _srodata), PTE_R |
    PTE_V);
12
13 // mapping other memory -|W|R|V
14 create_mapping((uint64_t *)swapper_pg_dir, _sdata,
    VA2PA(_sdata), PGROUNDUP(_edata - _sdata), PTE_R | PTE_W |
    PTE_V);
15 create_mapping((uint64_t *)swapper_pg_dir, _sbss,
    VA2PA(_sbss), PGROUNDUP(_ebss - _sbss), PTE_R | PTE_W |
    PTE_V);
16 create_mapping((uint64_t *)swapper_pg_dir,
    PGROUNDUP((uint64_t)_ebss),
    VA2PA(PGROUNDUP((uint64_t)_ebss)), PHY_END -
    VA2PA(PGROUNDUP((uint64_t)_ebss)), PTE_R | PTE_W | PTE_V);
17
18 // set satp with swapper_pg_dir
19
20 // YOUR CODE HERE
21
22 set_pgtbl((uint64_t *)swapper_pg_dir);
23
24 return;
25 }

```

我们在 create_mapping 中检测 va, pa, size 是否 4k 对齐, 然后计算页面的数量, 调用 walk 来得到页表项的地址, 然后设置页表项的对应的 PPN 和 flag.

```

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      if (sz % PGSIZE != 0)
6      {
7          printk("create_mapping: size is not page aligned\n");
8      }
9      if (va % PGSIZE != 0)
10     {
11         printk("create_mapping: va is not page aligned\n");
12     }
13     if (pa % PGSIZE != 0)
14     {
15         printk("create_mapping: pa is not page aligned\n");
16     }
17     int pgcnt = sz / PGSIZE;
18     int level = 3;
19     for (int i = 0; i < pgcnt; i++)
20     {
21         pte_t *pte = walk((pagetable_ptr_t)pgtbl, va + i *
22             PGSIZE, 1, &level);
23         if (pte == 0)
24         {
25             printk("create_mapping: walk failed\n");
26         }
27         *pte = ADDR2PTE(pa + i * PGSIZE, perm, level);
28     }

```

walk 函数用于实现自动的进行页表项的查找和页表的分配。walk 还传入了 level 和 alloc 参数, 用于更加精细的控制 walk 的行为。这里尤其要注意的是, mm 模块的分配和释放均采用预计内核空间的虚拟地址, 所以在 walk 中需要进行虚拟地址和物理地址的转换。

```

1      /*
2      If return != 0, walk returns the address of the PTE for a given
       virtual address.
3      If alloc != 0, walk allocates a new page table page for a given
       virtual address if one doesn't exist.

```

```

4   If level == 0, don't return level, but if alloc != 0, return 0.
5   If *level == 0, return the level of the page table entry, but if
    alloc != 0, return 0.
6   If *level == 1 or 2 or 3, return the address of the PTE for a
    given virtual address and the given level.
7   */
8   pte_t *walk(pagetable_ptr_t pagetable, uint64_t va, int alloc,
    int *level)
9   {
10      int set_level = !(level == 0 || *level == 0);
11      if (!set_level && alloc != 0)
12      {
13          return 0;
14      }
15      if (level != 0 && (*level < 0 || *level > 3))
16      {
17          return 0;
18      }
19      pte_t *pte;
20      int current_level = 1;
21      pagetable_ptr_t current_pagetable = pagetable;
22      for (; current_level < (set_level ? *level : 3);
    current_level++)
23      {
24          pte = &((*current_pagetable)[VPN(va, current_level)]);
25          if (*pte & PTE_V)
26          {
27              if ((*pte & PTE_R) || (*pte & PTE_W) || (*pte & PTE_X))
28              {
29                  break;
30              }
31              current_pagetable =
    (pagetable_ptr_t)PA2VA(PTE2ADDR(*pte, 3));
32          }
33          else if (alloc == 0)

```

```

34         {
35             return 0;
36         }
37         else
38         {
39             pagetable_ptr_t new_pagetable =
40                 (pagetable_ptr_t)kalloc();
41             if (new_pagetable == 0)
42             {
43                 return 0;
44             }
45             memset(new_pagetable, 0, PGSIZE);
46             *pte =
47                 ADDR2PTE((uint64_t)(VA2PA((uint64_t)new_pagetable))
48                     , PTE_V, 3);
49             current_pagetable = new_pagetable;
50         }
51         // printk("1");
52         if (set_level && current_level != *level)
53         {
54             // printk("walk: level not match\n");
55             return 0;
56         }
57         if (!set_level && level != 0)
58         {
59             *level = current_level;
60         }
61         return &(((current_pagetable)[VPN(va, current_level)]));
62     }

```

我们还需要完善 mm 模块, 因为先前 mm 模块是采用物理地址进行分配和释放的, 现在我们需要采用预计内核空间的虚拟地址进行分配和释放. 我们需要修改 mm_init 中释放空间的范围.

```

1     void mm_init(void) {

```

```

2      kfreerange(PGROUNDUP((uint64_t)_ekernel), (char *) (PHY_SIZE +
      VM_START));
3      printk("...mm_init done!\n");
4      return;
5  }

```

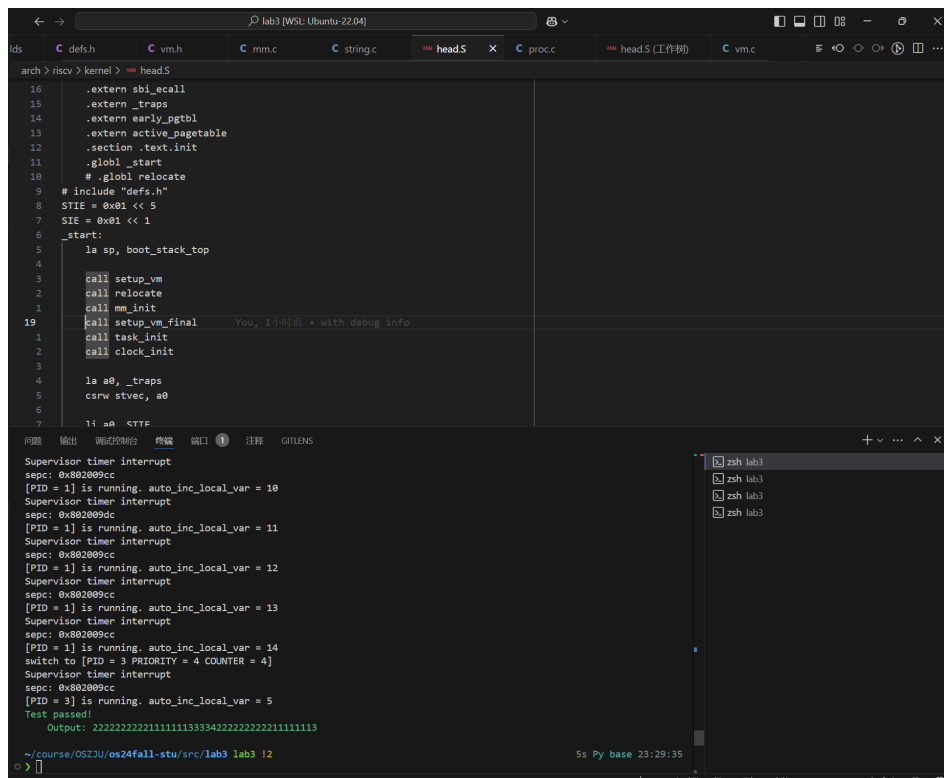
现在我们已经将未来的页表配置完毕, 现在我们需要实现 `set_pgtbl` 函数, 并在 `setup_vm_final` 中调用, 该函数的作用是将页表的地址写入 `satp` 寄存器, 从而使得页表生效. 注意传入的参数是虚拟地址, 所以我们需要进行虚拟地址和物理地址的转换, 才能将页表的地址写入 `satp` 寄存器.

```

1  // the pgtbl is VA
2  void set_pgtbl(uint64_t *pgtbl)
3  {
4      uint64_t satp = VA2PA((uint64_t)pgtbl) >> 12;
5      asm volatile(
6          "fence\n"
7          "fence.i\n"
8          "sfence.vma zero, zero\n"
9          "csrr t1, satp \n"
10         "li t2, 0xfffff00000000000 \n"
11         "li t3, 0x00000fffffffffff \n"
12         "and t1, t1, t2 \n"
13         "and t3, %0, t3 \n"
14         "or t1, t1, t3 \n"
15         "csrw satp, t1 \n"
16         "fence\n"
17         "fence.i\n"
18         "sfence.vma zero, zero\n"
19         :
20         : "r"(satp)
21         : "memory");
22  }

```

启动内核, 正常运行, 通过 lab2 的测试.



```
arch > riscv > kernel > head.S
16 .extern sbi_ecall
15 .extern _traps
14 .extern early_pgtbl
13 .extern active_pagetable
12 .section .text.init
11 .globl _start
10 # .globl relocate
9 # include "defs.h"
8 STIE = 0x01 << 5
7 SIE = 0x01 << 1
6 _start:
5 la sp, boot_stack_top
4
3 call setup_vm
2 call relocate
1 call mm_init
19 [call setup_vm_final You, 1小时 前 • with debug info]
1 call task_init
2 call clock_init
3
4 la a0, _traps
5 csw stvec, a0
6
7 li a0, STIE

问题 输出 调试控制台 终端 窗口 1 注释 GDB/LLDB
Supervisor timer interrupt
sepc: 0x802009cc
[PID = 1] is running. auto_inc_local_var = 10
Supervisor timer interrupt
sepc: 0x802009cc
[PID = 1] is running. auto_inc_local_var = 11
Supervisor timer interrupt
sepc: 0x802009cc
[PID = 1] is running. auto_inc_local_var = 12
Supervisor timer interrupt
sepc: 0x802009cc
[PID = 1] is running. auto_inc_local_var = 13
Supervisor timer interrupt
sepc: 0x802009cc
[PID = 1] is running. auto_inc_local_var = 14
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
Supervisor timer interrupt
sepc: 0x802009cc
[PID = 3] is running. auto_inc_local_var = 5
Test passed!
Output: 2222222221111111333422222222222222211111113
~/course/OS2JU/os24fall-stu/src/lab3 lab3 |2
$ > []
5s Py base 23:29:35
```

图 9: setup_vm_final 阶段 lab2 测试

3 实验结果与分析

我们在编写完成后, 运行内核, 进行 lab2 的测试, 结果如下:

```

switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 1
Supervisor timer interrupt
sepc: 0xffffffff0002009e8
[PID = 3] is running. auto_inc_local_var = 2
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 3] is running. auto_inc_local_var = 3
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 3] is running. auto_inc_local_var = 4
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
[PID = 4] is running. auto_inc_local_var = 1
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
Supervisor timer interrupt
sepc: 0xffffffff0002009e8
[PID = 2] is running. auto_inc_local_var = 11
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 12
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 13
Supervisor timer interrupt
sepc: 0xffffffff0002009e8
[PID = 2] is running. auto_inc_local_var = 14
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 15
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 16
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 17
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 18
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 19
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 2] is running. auto_inc_local_var = 20
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 1] is running. auto_inc_local_var = 8
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 1] is running. auto_inc_local_var = 9
Supervisor timer interrupt
sepc: 0xffffffff0002009e8
[PID = 1] is running. auto_inc_local_var = 10
Supervisor timer interrupt
sepc: 0xffffffff0002009e8
[PID = 1] is running. auto_inc_local_var = 11
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 1] is running. auto_inc_local_var = 12
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 1] is running. auto_inc_local_var = 13
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 1] is running. auto_inc_local_var = 14
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
Supervisor timer interrupt
sepc: 0xffffffff0002009d8
[PID = 3] is running. auto_inc_local_var = 5
Test passed!
Output: 22222222221111113333422222222211111113

~/c/OSZJU/os24fall-stu/src/lab3 lab3 !2 5s Py base 00:12:51
> 

```

图 10: lab2 测试结果

可以看到我实现的虚拟内存管理功能正常运行, 通过了所有测试用例.

4 遇到的问题及解决方法

我遇到过的问题主要是在运行时会莫名其妙的陷入循环, 导致无法正常运行. 后来经过检查, 发现主要是因为是在设置页表时, 和配置地址时有多处的物理地址和虚拟地址的混淆, 导致了地址映射错误.

5 总结与心得

虚拟内存实验是一个比较复杂的实验, 需要对 RISC-V 的页表机制有一定的了解, 同时也需要对操作系统的内存管理有一定的了解. 通过本次实验, 我学会了如何使用 RISC-V 的页表机制来实现虚拟内存, 并且了解了虚拟内存的相关知识, 对操作系统的内存管理有了更深入的了解.

6 思考题

6.1 验证.text,.rodata 段的属性是否成功设置, 给出截图.

我先编写了一个简单的函数 show_pgtbl 来查看页表的内容.

```
1 void show_pgtbl(uint64_t *pgtbl)
2 {
3     for(int i=0;i<512;i++)
4     {
5         if(pgtbl[i]&PTE_V)
6         {
7             printk("VPN: %p\n",i);
8             printk("PTE: %p\n",pgtbl[i]);
9             if(!(pgtbl[i]&PTE_R)&&!(pgtbl[i]&PTE_W)
10                &&!(pgtbl[i]&PTE_X))
11             {
12                 pagetable_ptr_t current_pagetable1 =
13                     (pagetable_ptr_t)PA2VA(PTE2ADDR(pgtbl[i],3));
14                 for(int j=0;j<512;j++)
15                 {
16                     if((*current_pagetable1)[j]&PTE_V)
```

```

16         {
17             printk("\tVPN: %p\n",j);
18             printk("\tPTE:
19                 %p\n",(*current_pagetable1)[j]);
20             if(!(pgtbl[i]&PTE_R)&&!(pgtbl[i]&PTE_W)
21                 &&!(pgtbl[i]&PTE_X))
22             {
23                 pagetable_ptr_t current_pagetable2 =
24                     (pagetable_ptr_t)
25                     PA2VA(PTE2ADDR((*current_pagetable1)[j],
26                                     3));
27                 for(int k=0;k<512;k++)
28                 {
29                     if((*current_pagetable2)[k]&PTE_V)
30                     {
31                         printk("\t\tVPN: %p\n",k);
32                         printk("\t\tPTE: %p\n",
33                             (*current_pagetable2)[k]);
34                     }
35                 }
36             }
37         }
38     }
39 }

```

.text 段的权限是 R|X, .rodata 段的权限是 R. 我们在实际中测试了一下.text 段和.rodata 段的写入是否会触发中断. 我们还在输出中测试了读取, 而.text 段的执行权限因为程序一直在执行无需验证.

```

1     #include "printk.h"
2     #include "vm.h"
3     #include "../arch/riscv/include/defs.h"
4     extern void test();

```

```

5  extern char _stext[], _etext[], _srodata[], _erodata[];
6  extern unsigned long swapper_pg_dir[512]
   __attribute__((__aligned__(0x1000)));
7  extern void show_pgtbl(uint64_t *pgtbl);
8  int start_kernel() {
9      printk(".text: %p ~ %p\n", _stext, _etext);
10     printk(".rodata: %p ~ %p\n", _srodata, _erodata);
11     char *p = _stext;
12     char *q = _srodata;
13     printk("p: %p, belongs to .text, its value is %p\n", p, *p);
14     printk("q: %p, belongs to .rodata, its value is %p\n", q, *q);
15     printk("write to p, expected to be store-page-fault\n");
16     *p = *q;
17     printk("write to q, expected to be store-page-fault\n");
18     *q = *p;
19     show_pgtbl(swapper_pg_dir);
20     return 0;
21 }

```

验证结果如下图所示:

```

...mm_init done!
...task_init done!
.text: 0xffffffff00020000 ~ 0xffffffff000202ed0
.rodata: 0xffffffff000203000 ~ 0xffffffff000203620
p: 0xffffffff00020000, belongs to .text, its value is 0x17
q: 0xffffffff000203000, belongs to .rodata, its value is 0x2e
write to p, expected to be store-page-fault
Store/AMO page fault
sepc: 0xffffffff000201e18
write to q, expected to be store-page-fault
Store/AMO page fault
sepc: 0xffffffff000201e34
VPN: 0x180
PTE: 0x21fffc01
    VPN: 0x1
    PTE: 0x21fff801
        VPN: 0x0
        PTE: 0x2008004b
        VPN: 0x1
        PTE: 0x2008044b
        VPN: 0x2
        PTE: 0x2008084b
        VPN: 0x3
        PTE: 0x20080c43
        VPN: 0x4
        PTE: 0x20081047
        VPN: 0x5
        PTE: 0x200814c7
        VPN: 0x6
        PTE: 0x200818c7
        VPN: 0x7
        PTE: 0x20081c07
        VPN: 0x8

```

图 11: 验证.text,.rodata 段的属性是否成功设置

我可以看到实际权限执行符合预期, 且页表中第 1-3 个 3 级页表属于.text 段, 属性为 R|X|V, 第 4 个 3 级页表属于.rodata 段, 属性为 R|V, 符合预期.

6.2 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. 你需要回答以下问题:

a. 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

会导致内核无法正常取指令, 这是因为代码执行切换页表指令时, 必然使用的是物理地址或者是旧页表的虚拟地址, 而在之后要将指令流从原来的区域切换到新的区域, 需要再取一条指令, 但是此时新的页表已经生效, 无法再取到原来的指令. 一般来说, 页表切换时, 新旧页表需要有一般分映射是相同的 (采用物理地址时, 等同于采用了一个等值映射的页表), 所以中间页表需要有一段等值映射.

b. 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

内核首先禁用 S 模式中断, 加载 gp, 禁用浮点单元, 选择 (确定) 一个用于运行系统内核启动代码的处理器核心, 按照需求清空 BSS 段, 保存核心 ID 和 DTB 的物理地址, 配置系统栈.

然后调用 `setup_vm` 进行页表的配置, 在 `setup_vm` 函数中, 系统配置了物理虚拟地址之间的偏移量和物理地址开始 (基地址) 的 PPN, 随后更具是否有中间页表配置了顶级页表 `trampoline_pg_dir` 和 `swapper_pg_dir`. 其中 `trampoline_pg_dir` 只有内核虚拟地址空间一页映射, 对应的是内核虚拟基地址页 (对应的大概是 `_start`, `relocate` 等系统启动代码的部分) 映射到物理基地址页, 而 `swapper_pg_dir` 映射了整块内核虚拟空间到物理基地址开始的空间, 还映射了一块中间页 `fixmap_pte`.

之后再调用 `relocate` 函数, 进行页表的切换, 首先将 `ra` 加上偏移量, 将 `stvec` 设置为页表切换指令的下一条指令的虚拟地址, 配置好 `swapper_pg_dir` 和 `trampoline_pg_dir` 对应的 `satp`, 之后先切换到 `trampoline_pg_dir`, 之后再切换到 `swapper_pg_dir`, 恢复 `stvec`, 这样就完成了页表的切换.

最后恢复 C 语言执行环境 (栈等), 最后启用系统内核.

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

因为 Linux 在正常情况下不是使用跳转指令进行指令流从物理空间到虚拟空间的转换, 而是使用了中断触发过程中产生的跳转. 在中断触发过程中, 会将 `stvec` 寄存器

设置为页表切换指令的下一条指令的虚拟地址, 并将 ra 寄存器设置为虚拟地址, 之后在页表切换后, 由于正常情况下没有设置物理地址的等值映射, 会导致无法继续执行, 触发 Instruction access fault, 进入异常处理流程, 进而跳转到 stvec 寄存器指向的虚拟地址, 从而完成了从物理地址到虚拟地址的跳转, 并且在函数返回之后, 也是返回 ra 对应的虚拟地址.

d.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

trampoline_pg_dir 只有内核虚拟地址空间一页映射, 对应的是内核虚拟基地址页 (对应的大概是 __start, relocate 等系统启动代码的部分) 映射到物理基地址页, 而 swapper_pg_dir 映射了整块内核虚拟空间到物理基地址开始的空间, 还映射了一块中间页 fixmap_pte. trampoline_pg_dir 是 swapper_pg_dir 的子集. 这两个顶级页表的切换位置如下:

```
1      /* Compute satp for kernel page tables, but don't load it yet */
2      la a2, swapper_pg_dir
3      srl a2, a2, PAGE_SHIFT
4      li a1, SATP_MODE
5      or a2, a2, a1
6
7      /*
8       * Load trampoline page directory, which will cause us to trap to
9       * stvec if VA != PA, or simply fall through if VA == PA. We
10       * need a
11       * full fence here because setup_vm() just wrote these PTEs and
12       * we need
13       * to ensure the new translations are in use.
14       */
15      la a0, trampoline_pg_dir
16      srl a0, a0, PAGE_SHIFT
17      or a0, a0, a1
18      sfence.vma
19      csrwr CSR_SATP, a0 <----- trampoline_pg_dir
20
21      .align 2
22  1:
23
24      /* Set trap vector to spin forever to help debug */
```



```

21     la a0, .Lsecondary_park
22     csrwr CSR_STVEC, a0
23
24     /* Reload the global pointer */
25 .option push
26 .option norelax
27     la gp, __global_pointer$
28 .option pop
29
30     /*
31      * Switch to kernel page tables. A full fence is necessary in
32      * order to
33      * avoid using the trampoline translations, which are only
34      * correct for
35      * the first superpage. Fetching the fence is guaranteed to work
36      * because that first superpage is translated the same way.
37      */
38     csrwr CSR_SATP, a2 <----- swapper_pg_dir
39     sfence.vma

```

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

我们首先关闭了等值映射, 修改了 `setup_vm` 函数如下:

```

1     void setup_vm(void)
2     {
3         /*
4          * 1. 由于是进行 1GiB 的映射, 这里不需要使用多级页表
5          * 2. 将 va 的 64bit 作为如下划分: | high bit | 9 bit | 30
6             bit |
7          * high bit 可以忽略
8          * 中间 9 bit 作为 early_pgtbl 的 index
9          * 低 30 bit 作为页内偏移, 这里注意到 30 = 9 + 9 +
10             12, 即我们只使用根页表, 根页表的每个 entry 都对应 1GiB 的区域
11          * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
12          */

```

```

11     memset(early_pgtbl, 0, PGSIZE);
12     pte_t *pte;
13     int level = 1;
14     pte = (pte_t *)&early_pgtbl[VPN(VM_START, level)];
15     *pte = ADDR2PTE(PHY_START, PTE_V | PTE_R | PTE_W | PTE_X, 1);
16     // pte = (pte_t *)&early_pgtbl[VPN(PHY_START, level)];
17     // *pte = ADDR2PTE(PHY_START, PTE_V | PTE_R | PTE_W | PTE_X,
18         1);
19 }

```

然后在修改 `_start` 函数, 使得中断初始化和启用在 `setup_vm` 直接完成:

```

1  _start:
2  la sp, boot_stack_top
3
4  la a0, _traps
5  csrw stvec, a0
6
7  li a0, STIE
8  csrs sie, a0
9
10
11 li a0, SIE
12 csrs sstatus, a0
13
14 call setup_vm
15 call relocate
16 call mm_init
17 call setup_vm_final
18 call task_init
19 call clock_init
20
21 call start_kernel

```

最后我们更改 `relocate` 函数, 在切换页表前设置 `stvec` 指向切换页表的下一条指令, 并马上恢复 `stvec` 到原有值对应的虚拟地址, 并启用中断.

```

1  relocate:
2  # set ra = ra + PA2VA_OFFSET
3  # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
4
5  li t1, PA2VA_OFFSET
6  add ra, ra, t1
7  add sp, sp, t1
8
9  # set satp with early_pgtbl
10
11  la t0, early_pgtbl # PPN
12  srli t1, t0, 12
13  li t2, 8 # MODE
14  slli t2, t2, 60
15  or t1, t1, t2
16  li t2, 0xf0000fffffffffff # ASID
17  and t1, t1, t2
18
19  li t5, PA2VA_OFFSET
20  csrr t4, stvec
21  la t3, vm_set_vec
22  add t3, t3, t5
23  add t4, t4, t5
24
25  csrw stvec, t3
26
27  # flush d.cache
28  fence
29
30  # flush tlb
31  sfence.vma zero, zero
32
33  # flush icache
34  fence.i
35

```

```
36     csrw satp, t1
37
38     vm_set_vec:
39     csrw stvec, t4
40
41     li a0, SIE
42     csrs sstatus, a0
43
44
45     # flush d.cache
46     fence
47
48     # flush tlb
49     sfence.vma zero, zero
50
51     # flush icache
52     fence.i
53
54     ret
```

运行 lab2 得测试, 结果如下图:

```
arch > riscv > kernel > head.S
51 relocate:
27 | csrw stvec, t3
26 |
25 | # flush d.cache
24 | fence
23 |
22 | # flush tlb
21 | sfence.vma zero, zero
20 |
19 | # flush icache
18 | fence.i
17 |
16 | csrw satp, t1
15 |
14 | vm_set_vec:
13 | csrw stvec, t4
12 |
11 | li a0, SIE
10 | csrs sstatus, a0
9 |
8 |
7 | # flush d.cache
6 | fence
5 |
4 | # flush tlb

问题 输出 调试控制台 终端 窗口 注册 GITHUBS
sepc: 0xffffffff00200a04
[PID = 1] is running. auto_inc_local_var = 10
Supervisor timer interrupt
sepc: 0xffffffff00200a14
[PID = 1] is running. auto_inc_local_var = 11
Supervisor timer interrupt
sepc: 0xffffffff00200a04
[PID = 1] is running. auto_inc_local_var = 12
Supervisor timer interrupt
sepc: 0xffffffff00200a04
[PID = 1] is running. auto_inc_local_var = 13
Supervisor timer interrupt
sepc: 0xffffffff00200a04
[PID = 1] is running. auto_inc_local_var = 14
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
Supervisor timer interrupt
sepc: 0xffffffff00200a14
[PID = 3] is running. auto_inc_local_var = 5
Test passed!
Output: 222222222111111133334222222222211111113
~/course/OS23U/os24fall-stu/src/lab3 lab3 i2
55 Py base 17:41:32
```

图 12: 不做等值映射的结果