浙江大學

《操作系统原理与实践》
实验报告

实验名称	：	Lab 3: RV64 虚拟内存管理
姓　　名	：	王晓宇
学　　号	：	3220104364
电子邮箱	：	3220104364@zju.edu.cn
联系电话	：	19550222634
授课教师	：	申文博
助　　教	：	陈淦豪&王鹤翔&许昊瑞

2024 年 11 月 17 日

Lab 3: RV64 虚拟内存管理

Lab 3: RV64 虚拟内存管理

1 实验内容及简要原理介绍

- 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

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**。

2 实验具体过程与代码实现

2.1 准备工程_4.1

已同步Lab2代码

- 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)
```

- 同步 `vmlinux.lds` 代码，并按照以下步骤将这些文件正确放置：

```
.
└── arch
    └── riscv
        └── kernel
            └── vmlinux.lds
```

2.2 关于PIE_4.2

在 Makefile 的 `CF` 中加一个 `-fno-pie`:

```
CF      :=  -march=$(ISA) -mabi=$(ABI) -mcmodel=medany  -fno-pie -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g

```

2.3 开启虚拟内存映射_4.3

2.3.1 setup_vm 的实现

- vm.h

```
#ifndef __VM_H__
#define __VM_H__
#include "stdint.h"
#include "defs.h"
#include "printk.h"
#include "string.h"
#include "mm.h"

void setup_vm();

#endif
```

- vm.c

```
#include <vm.h>
/* early_pgtbl: 用于 setup_vm 进行 1GiB 的映射 */
uint64_t early_pgtbl[512] __attribute__((__aligned__(0x1000)));
void setup_vm() {
    /*
```

```
     * 1. 由于是进行 1GiB 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
     *     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 1GiB 的区域
     * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    **/
    memset(early_pgtbl, 0x0, PGSIZE);
    // first mapping
    early_pgtbl[(uint64_t)PHY_START>>30] = ((uint64_t)(PHY_START>>30)<<28) | (uint64_t)0xf;
    //second mapping
    early_pgtbl[((uint64_t)(PHY_START+PA2VA_OFFSET)>> 30) & 0x1ff] = ((uint64_t)(PHY_START>>30)<<28) | (uint64_t)0xf;
}
```

- head.S

```
    .extern start_kernel
    .extern _traps
    .extern sbi_set_timer
    .extern task_init
    .extern mm_init

    .extern setup_vm
    .extern early_pgtbl
    .extern setup_vm_final

    .section .text.init
    .globl _start
_start:
    la sp, boot_stack_top

    call setup_vm
    call relocate
    ...
```

```



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

    li t0,0xffffffdf80000000
    add ra, ra, t0
    add sp, sp, t0

    # need a fence to ensure the new translations are in use
    sfence.vma zero, zero

    # set satp with early_pgtbl

    la t1, early_pgtbl
    srli t1, t1, 12
    li t0, 0x8
    slli t0, t0, 30
    slli t0, t0, 30
    or t0, t0, t1
    csrw satp, t0

    ret
    .section .bss.stack
    .globl boot_stack
boot_stack:
    ...
```

2.3.2 setup_vm_final 的实现

- vm.h

```
#ifndef __VM_H__
#define __VM_H__
#include "stdint.h"
#include "defs.h"

```

```c
#include "printk.h"
#include "string.h"
#include "mm.h"



void setup_vm();

void setup_vm_final();

void create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, uint64_t perm);


#endif
```

- vm.c

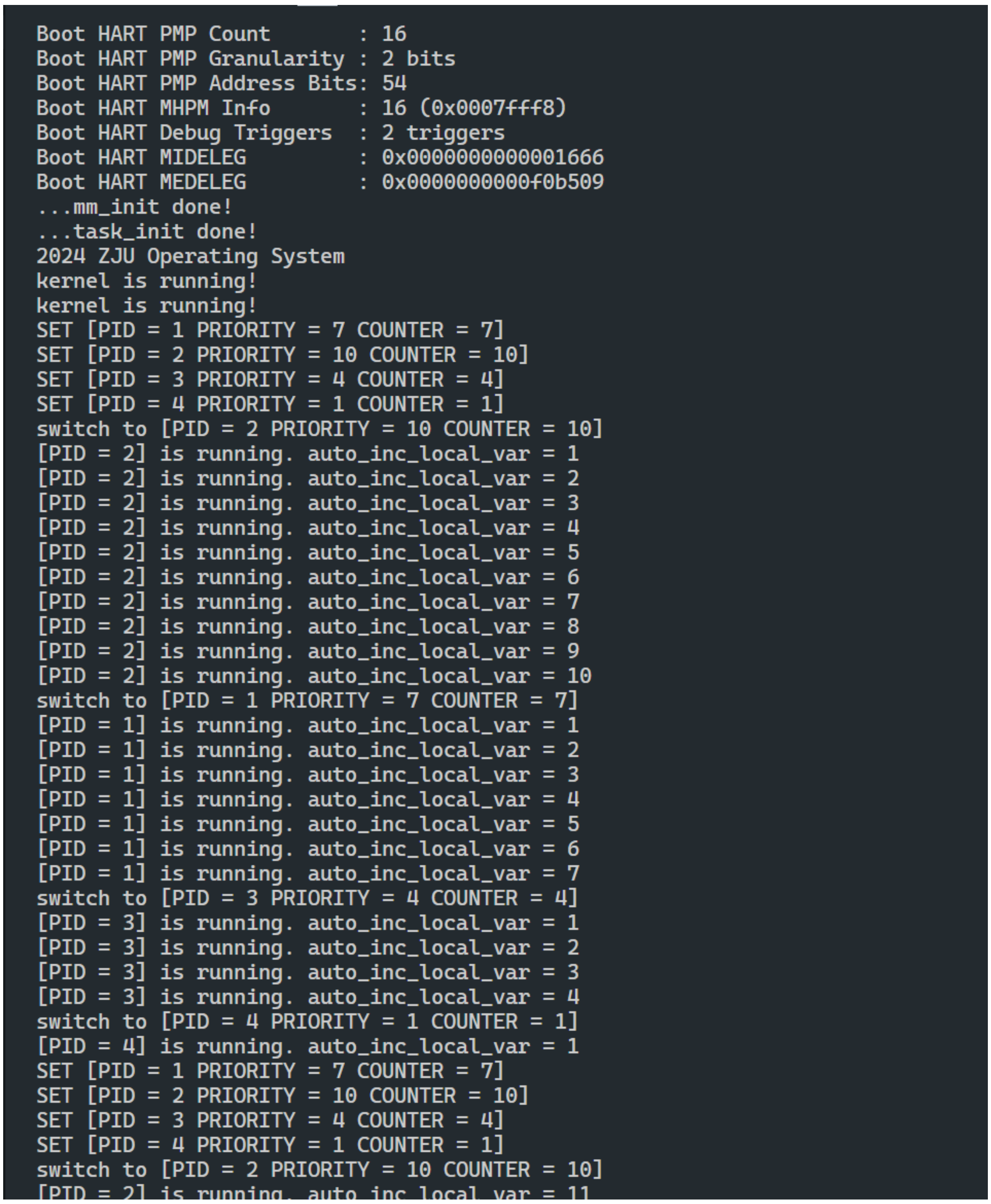
```c
#include <vm.h>
/* early_pgtbl: 用于 setup_vm 进行 1GiB 的映射 */
uint64_t early_pgtbl[512] __attribute__((__aligned__(0x1000)));
void setup_vm() {
    memset(early_pgtbl, 0x0, PGSIZE);
    // first mapping
    early_pgtbl[(uint64_t)PHY_START>>30] = ((uint64_t)(PHY_START>>30)<<28) | (uint64_t)0xf;
    //second mapping
    early_pgtbl[((uint64_t)(PHY_START+PA2VA_OFFSET)>> 30) & 0x1ff] = ((uint64_t)(PHY_START>>30)<<28) | (uint64_t)0xf;
}
/* swapper_pg_dir: kernel pagetable 根目录， 在 setup_vm_final 进行映射 */
uint64_t swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
extern uint64_t* _stext;
extern uint64_t* _srodata;
extern uint64_t* _sdata;
extern uint64_t* _etext;
extern uint64_t* _erodata;
extern uint64_t* _edata;

/* 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_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, uint64_t perm) {
   uint64_t va_end = va + sz;
   for(uint64_t temp_va = va;temp_va < va_end;temp_va+=PGSIZE){
        uint64_t vpn_2 = (temp_va >> 30) & 0x1ff;
        uint64_t vpn_1 = (temp_va >> 21) & 0x1ff;
        uint64_t vpn_0 = (temp_va >> 12) & 0x1ff;
        uint64_t pte_2 = pgtbl[vpn_2];
        uint64_t pte_1 ;
        uint64_t* pgtabl_1;
        uint64_t* pgtabl_0;
        if((pte_2 & 0x1) == 0x1){
            //second level page table exist
            pgtabl_1 = (uint64_t*)(((pte_2 & 0x003ffffffffffc00)<<2)+PA2VA_OFFSET);
        }else{
            pgtabl_1 = (uint64_t*)((((uint64_t)kalloc()-PA2VA_OFFSET)>>12)<<10);
            pgtbl[vpn_2] = (uint64_t)pgtabl_1 | 0x1;
            pgtabl_1 = (uint64_t*)((((uint64_t)pgtabl_1>>10)<<12) + PA2VA_OFFSET);
        }
        pte_1 = pgtabl_1[vpn_1];
        if((pte_1 & 0x1) == 0x1){
            //first level page table exist
            pgtabl_0 = (uint64_t*)(((pte_1 & 0x003ffffffffffc00)<<2) + PA2VA_OFFSET);
        }else{
            pgtabl_0 = (uint64_t*)((((uint64_t)kalloc()-PA2VA_OFFSET)>>12)<<10);
            pgtabl_1[vpn_1] = (uint64_t)pgtabl_0 | 0x1;
            pgtabl_0 = (uint64_t*)((((uint64_t)pgtabl_0>>10)<<12) + PA2VA_OFFSET);
        }
        pgtabl_0[vpn_0] = (((uint64_t)pa & 0x003ffffffffffc00)>>2) | perm;
        pa += PGSIZE;
   }
//    printk("finish vpn\n");
}

void setup_vm_final() {
    memset(swapper_pg_dir, 0x0, PGSIZE);

    // No OpenSBI mapping required

    // mapping kernel text X|-|R|V
    create_mapping(swapper_pg_dir, (uint64_t)&_stext, ((uint64_t)(&_stext)-PA2VA_OFFSET), ((uint64_t)(&_srodata)-(uint64_t)(&_stext)) , 0xb);

    // mapping kernel rodata -|-|R|V
    create_mapping(swapper_pg_dir, (uint64_t)&_srodata, ((uint64_t)(&_srodata)-PA2VA_OFFSET), ((uint64_t)(&_sdata)-(uint64_t)(&_srodata)) , 0x3);

    // mapping other memory -|W|R|V
    create_mapping(swapper_pg_dir, (uint64_t)&_sdata, ((uint64_t)(&_sdata)-PA2VA_OFFSET), (uint64_t)(PHY_SIZE - ((uint64_t)(&_sdata)-(uint64_t)(&_stext))) , 0x7);

    // set satp with swapper_pg_dir
    // printk("finish 3_mapping\n");
    uint64_t temp_satp = (((uint64_t)swapper_pg_dir-PA2VA_OFFSET)>>12) | (uint64_t)(0x8000000000000000);
    csr_write(satp,temp_satp);
    // flush TLB
    asm volatile("sfence.vma zero, zero");
    return;
}
```

- head.S

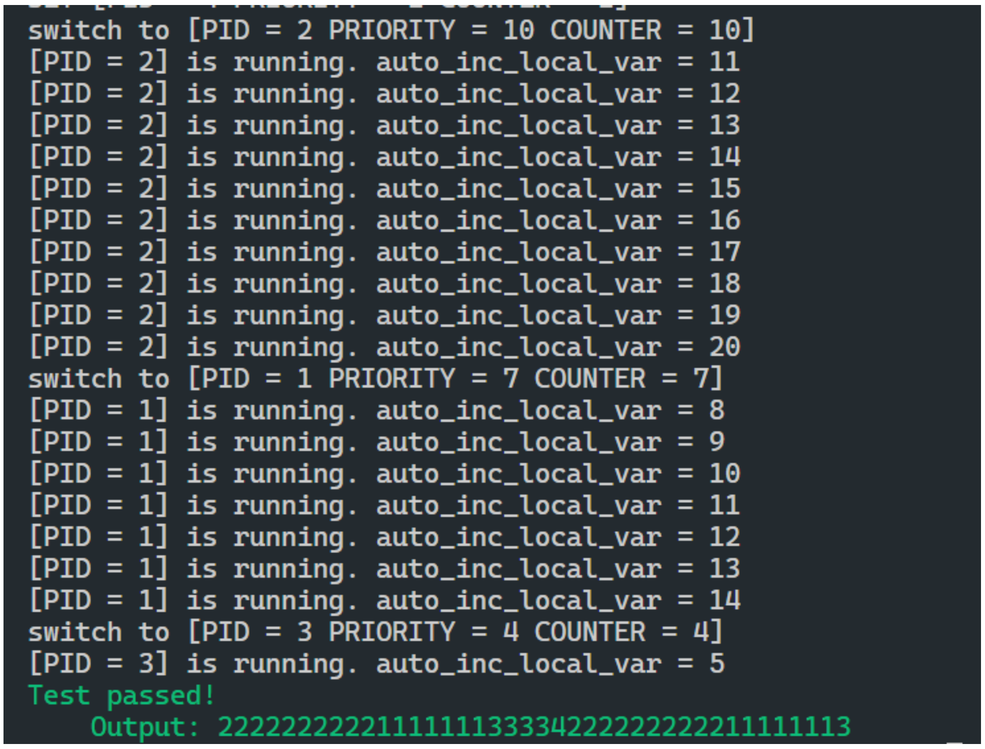
```asm
    .extern start_kernel
    .extern _traps
    .extern sbi_set_timer
    .extern task_init
    .extern mm_init

    .extern setup_vm
    .extern early_pgtbl
    .extern setup_vm_final

    .section .text.init
    .globl _start
_start:
    la sp, boot_stack_top

    call setup_vm
    call relocate

    call mm_init
    call setup_vm_final
    call task_init
    ...


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

    li t0,0xffffffdf80000000
    add ra, ra, t0
    add sp, sp, t0

    # need a fence to ensure the new translations are in use
    sfence.vma zero, zero

    # set satp with early_pgtbl

    la t1, early_pgtbl
    srli t1, t1, 12
    li t0, 0x8
    slli t0, t0, 30
    slli t0, t0, 30
    or t0, t0, t1
    csrw satp, t0

    ret
    .section .bss.stack
    .globl boot_stack
boot_stack:
    ...
```

3 实验结果与分析

> make TEST_SCHED=1 run输出



```
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 2 bits
Boot HART PMP Address Bits: 54
Boot HART MHPM Info       : 16 (0x0007fff8)
Boot HART Debug Triggers  : 2 triggers
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
...mm_init done!
...task_init done!
2024 ZJU Operating System
kernel is running!
kernel is running!
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 1
[PID = 2] is running. auto_inc_local_var = 2
[PID = 2] is running. auto_inc_local_var = 3
[PID = 2] is running. auto_inc_local_var = 4
[PID = 2] is running. auto_inc_local_var = 5
[PID = 2] is running. auto_inc_local_var = 6
[PID = 2] is running. auto_inc_local_var = 7
[PID = 2] is running. auto_inc_local_var = 8
[PID = 2] is running. auto_inc_local_var = 9
[PID = 2] is running. auto_inc_local_var = 10
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[PID = 1] is running. auto_inc_local_var = 1
[PID = 1] is running. auto_inc_local_var = 2
[PID = 1] is running. auto_inc_local_var = 3
[PID = 1] is running. auto_inc_local_var = 4
[PID = 1] is running. auto_inc_local_var = 5
[PID = 1] is running. auto_inc_local_var = 6
[PID = 1] is running. auto_inc_local_var = 7
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 1
[PID = 3] is running. auto_inc_local_var = 2
[PID = 3] is running. auto_inc_local_var = 3
[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
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 11
```



```
[illegible]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 11
[PID = 2] is running. auto_inc_local_var = 12
[PID = 2] is running. auto_inc_local_var = 13
[PID = 2] is running. auto_inc_local_var = 14
[PID = 2] is running. auto_inc_local_var = 15
[PID = 2] is running. auto_inc_local_var = 16
[PID = 2] is running. auto_inc_local_var = 17
[PID = 2] is running. auto_inc_local_var = 18
[PID = 2] is running. auto_inc_local_var = 19
[PID = 2] is running. auto_inc_local_var = 20
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[PID = 1] is running. auto_inc_local_var = 8
[PID = 1] is running. auto_inc_local_var = 9
[PID = 1] is running. auto_inc_local_var = 10
[PID = 1] is running. auto_inc_local_var = 11
[PID = 1] is running. auto_inc_local_var = 12
[PID = 1] is running. auto_inc_local_var = 13
[PID = 1] is running. auto_inc_local_var = 14
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 5
Test passed!
    Output: 2222222222111111133334222222222211111113
```

4 思考题与心得体会

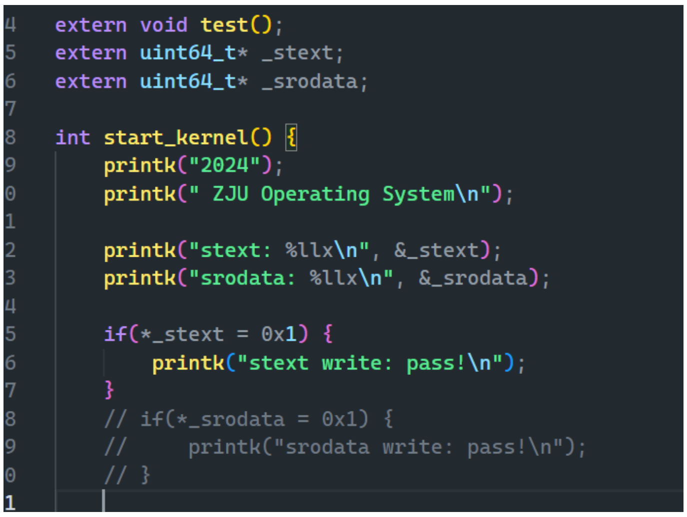
1. 验证 `.text`， `.rodata` 段的属性是否成功设置，给出截图。

我们从`setup_vm_final`中可以知道，对预期`.text`， `.rodata` 段的属性设置是：

> `.text`段可执行+不可写+可读+有效
>
> `.rodata` 段不可执行+不可写+可读+有效

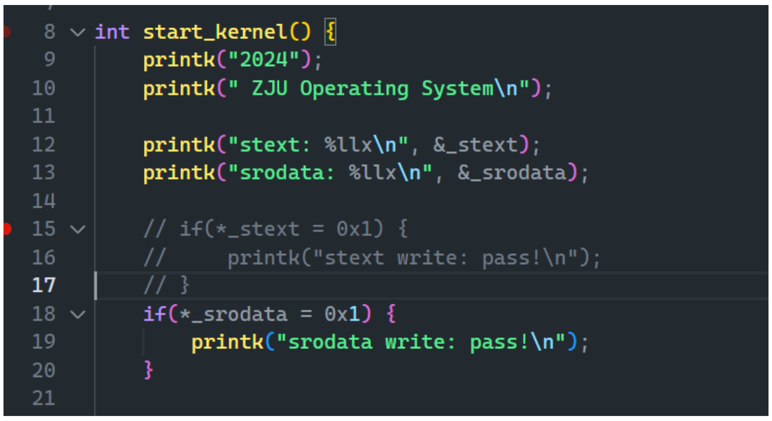
- 现在我们验证两段的读写性

  我们在`main.c`的`start_kernel`函数中作以下修改，使之不做进程调度算法，直接访问`.text`， `.rodata` 段：



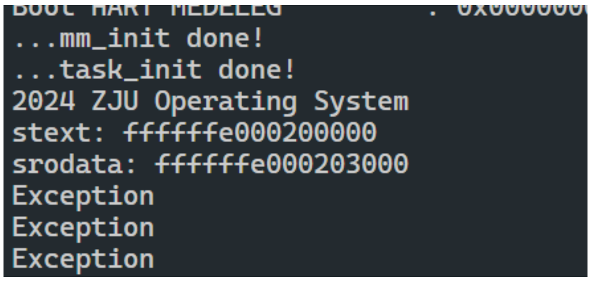
```c
extern void test();
extern uint64_t* _stext;
extern uint64_t* _srodata;

int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");

    printk("stext: %llx\n", &_stext);
    printk("srodata: %llx\n", &_srodata);

    if(*_stext = 0x1) {
        printk("stext write: pass!\n");
    }
    // if(*_srodata = 0x1) {
    //     printk("srodata write: pass!\n");
    // }
    
```

测试`.text`段的写入即上述代码，如测试`.rodata`则修改如下：

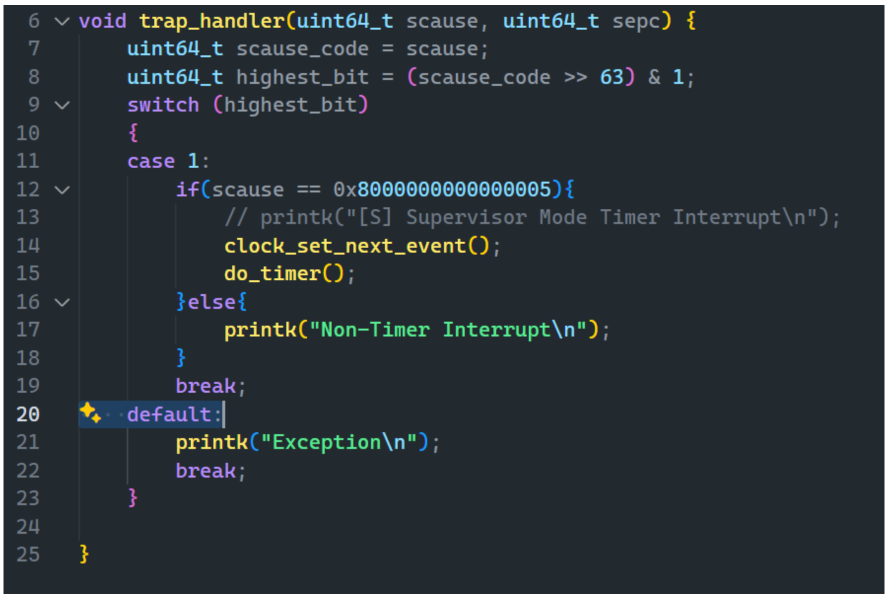


```c
int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");

    printk("stext: %llx\n", &_stext);
    printk("srodata: %llx\n", &_srodata);

    // if(*_stext = 0x1) {
    //     printk("stext write: pass!\n");
    // }
    if(*_srodata = 0x1) {
        printk("srodata write: pass!\n");
    }

```

两次断点测试结果均如下，可以看的出来，对两个段的读取命令有效，而对写入操作直接中断出错，是一次`Exception`

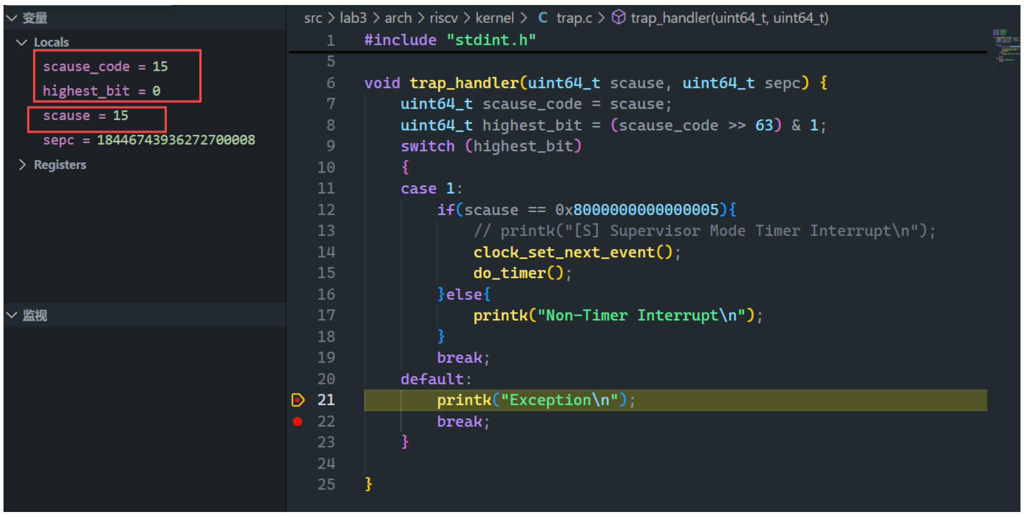


```
Boot HART MEDELEG         : 0x0000000
...mm_init done!
...task_init done!
2024 ZJU Operating System
stext: ffffffe000200000
srodata: ffffffe000203000
Exception
Exception
Exception
```



```c
void trap_handler(uint64_t scause, uint64_t sepc) {
    uint64_t scause_code = scause;
    uint64_t highest_bit = (scause_code >> 63) & 1;
    switch (highest_bit)
    {
    case 1:
        if(scause == 0x8000000000000005){
            // printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }else{
            printk("Non-Timer Interrupt\n");
        }
        break;
    default:
        printk("Exception\n");
        break;
    }

}
```

单步调试时我们可以找到`scause`的数值，从而观察异常出现的具体原因：

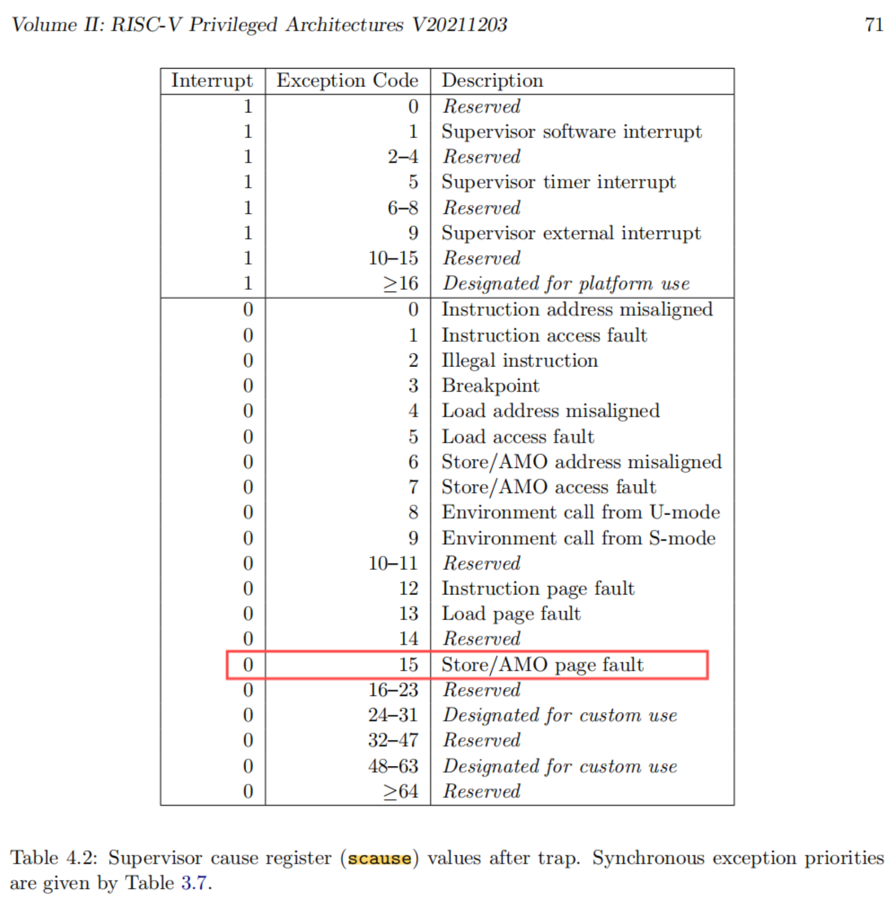


```
变量
  Locals
    scause_code = 15
    highest_bit = 0
    scause = 15
    sepc = 18446743936272700008
  Registers
监视
```

src > lab3 > arch > riscv > kernel > C trap.c > trap_handler(uint64_t, uint64_t)

```c
#include "stdint.h"

void trap_handler(uint64_t scause, uint64_t sepc) {
    uint64_t scause_code = scause;
    uint64_t highest_bit = (scause_code >> 63) & 1;
    switch (highest_bit)
    {
    case 1:
        if(scause == 0x8000000000000005){
            // printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }else{
            printk("Non-Timer Interrupt\n");
        }
        break;
    default:
        printk("Exception\n");
        break;
    }

}
```

查询特权级指令手册：



Volume II: RISC-V Privileged Architectures V20211203 71

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

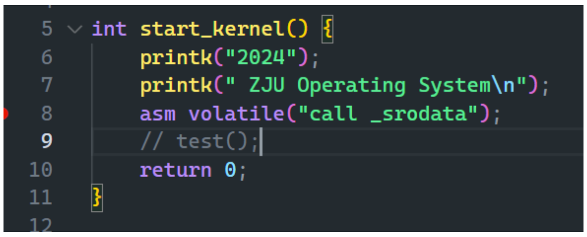
Table 4.2: Supervisor cause register (`scause`) values after trap. Synchronous exception priorities are given by Table 3.7.

首先`Interrupt`位即最高位为0对应表示异常(Exception)，`15`的数值对应`Store/AMO page fault`，当处理器尝试执行存储或原子内存操作时，如果目标地址所在的页面不存在或不可访问，由可读性我们可以知道该段是真实存在的，但是发生了存储异常说明这**两段代码均不可写**。

至此，我们完成了对这两段读写性的检查，属性设置正确。

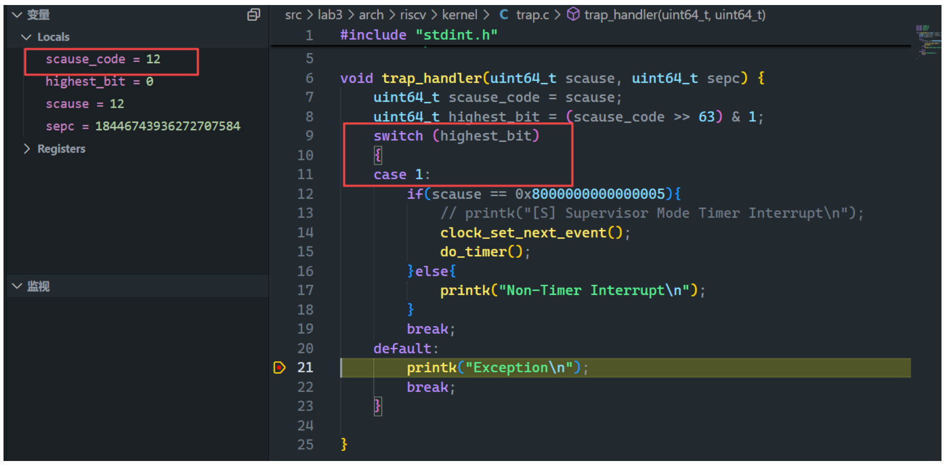
- **可执行性**：
    - `.text` 段用于存储程序的机器代码（即可执行指令），我们的程序存储在此段，程序能够正常运行实现进程切换功能即说明 `.text` 段是**可以被执行的**
    - `.rodata` 段用于存储只读数据，即在程序运行期间不会被修改的数据。这里的内存是不可以被执行的，我们只需要在`head.S`中做出跳转到该段所在地址的操作，如果可以被执行则说明属性设置错误

      我们在`main.c`的`start_kernel`函数中作以下修改，更改PC执行地址，使之执行至`.rodata`段：



```c
int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");
    asm volatile("call _srodata");
    // test();
    return 0;
}
```

单步调试发现在执行该内联汇编时跳入了`trap_handler`，此时显示`scause = 12`，并且此时`scause`的最高位为0：

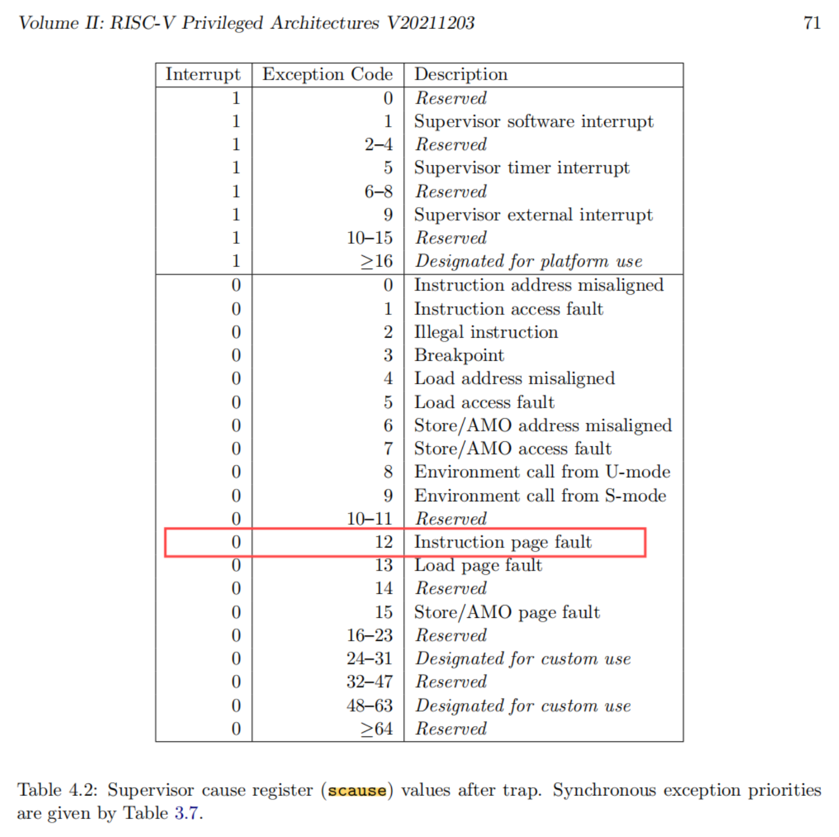


变量
Locals
Registers
监视
src > lab3 > arch > riscv > kernel > C trap.c > trap_handler(uint64_t, uint64_t)

```
scause_code = 12
highest_bit = 0
scause = 12
sepc = 18446743936272707584
```

```c
#include "stdint.h"

void trap_handler(uint64_t scause, uint64_t sepc) {
    uint64_t scause_code = scause;
    uint64_t highest_bit = (scause_code >> 63) & 1;
    switch (highest_bit)
    {
    case 1:
        if(scause == 0x8000000000000005){
            // printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }else{
            printk("Non-Timer Interrupt\n");
        }
        break;
    default:
        printk("Exception\n");
        break;
    }

}
```

查询特权级指令及寄存器手册得知，`Interrupt`位即最高位为0对应表示异常(Exception)，`12`的数值对应`Instruction page fault`



Volume II: RISC-V Privileged Architectures V20211203 71

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

Table 4.2: Supervisor cause register (**scause**) values after trap. Synchronous exception priorities are given by Table 3.7.

当处理器尝试从内存中获取指令时，如果该指令所在的页面不存在或不可访问，就会触发这种异常，但是这页是真实存在的，而且从上文读写性测试我们可以得知该页也是可以读取的，发生fault的原因便是该段**不能作为可执行段**去访问。

至此，我们完成了对这两段可执行性的检查，属性设置正确。

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：

- 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；

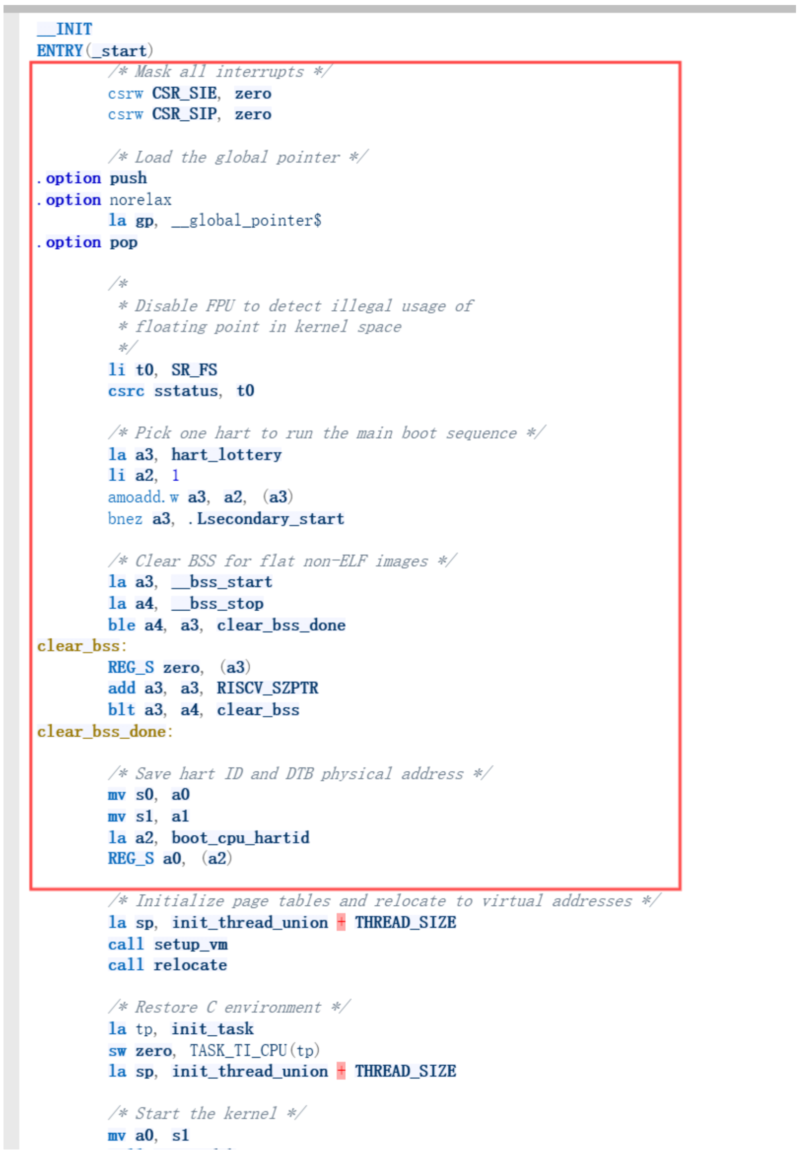
  会出现缺页错误无法访问下一条指令，导致程序中断或卡住。

  在不做等值映射时，当我们利用`csrw`指令设置satp之后即开启MMU功能，我们此时应该使用虚拟地址当作PC喂给MMU转化为物理地址执行。

  但是`csrw`指令此时的PC值+4是`csrw`的物理地址+4，得到的这个地址当作虚拟地址喂给MMU之后，如果没有等值映射，那这个就是缺页异常，找不到这个地址对应的物理地址，导致程序无法进行。

- 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；

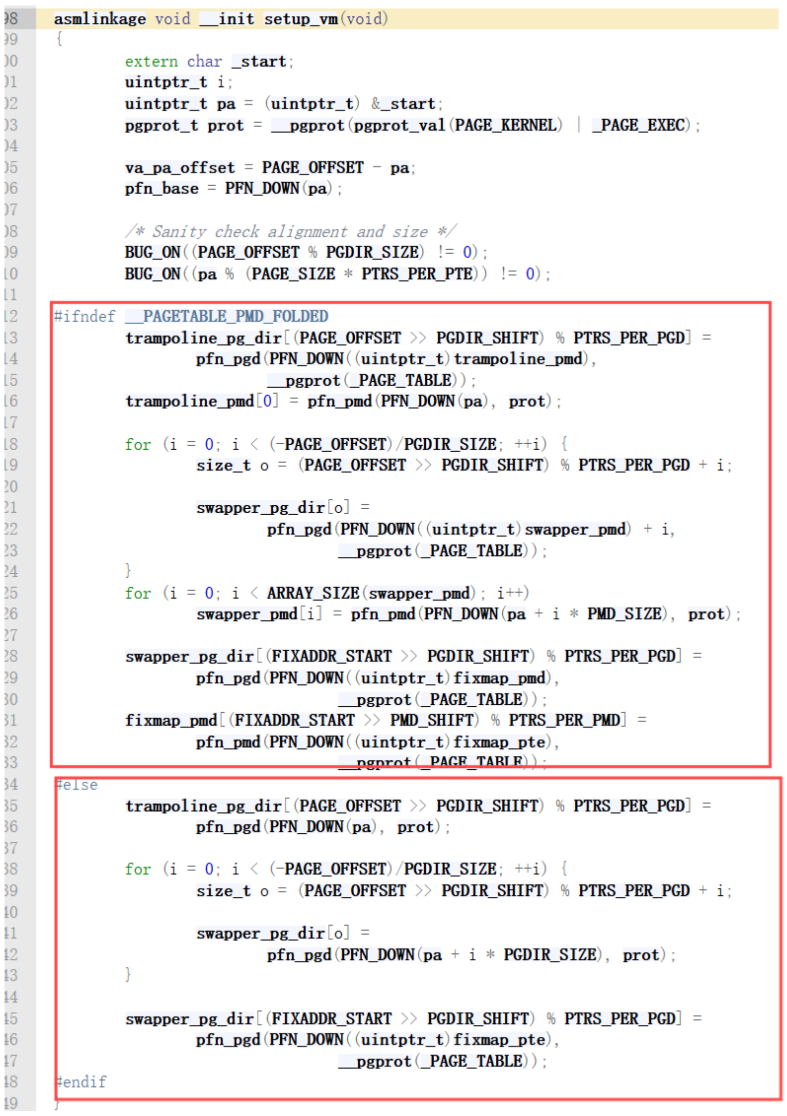
  > 以下截自Linux v5.2.21的_start入口



```
__INIT
ENTRY(_start)
        /* Mask all interrupts */
        csrw CSR_SIE, zero
        csrw CSR_SIP, zero

        /* Load the global pointer */
.option push
.option norelax
        la gp, __global_pointer$
.option pop

        /*
         * Disable FPU to detect illegal usage of
         * floating point in kernel space
         */
        li t0, SR_FS
        csrc sstatus, t0

        /* Pick one hart to run the main boot sequence */
        la a3, hart_lottery
        li a2, 1
        amoadd.w a3, a2, (a3)
        bnez a3, .Lsecondary_start

        /* Clear BSS for flat non-ELF images */
        la a3, __bss_start
        la a4, __bss_stop
        ble a4, a3, clear_bss_done
clear_bss:
        REG_S zero, (a3)
        add a3, a3, RISCV_SZPTR
        blt a3, a4, clear_bss
clear_bss_done:

        /* Save hart ID and DTB physical address */
        mv s0, a0
        mv s1, a1
        la a2, boot_cpu_hartid
        REG_S a0, (a2)

        /* Initialize page tables and relocate to virtual addresses */
        la sp, init_thread_union + THREAD_SIZE
        call setup_vm
        call relocate

        /* Restore C environment */
        la tp, init_task
        sw zero, TASK_TI_CPU(tp)
        la sp, init_thread_union + THREAD_SIZE

        /* Start the kernel */
        mv a0, s1
```

框起来的部分是启动前的一些初始化，建立两个页表前的准备工作，暂不考虑他们的作用，大概是内核启动的选核、指针设置等等工作。

我们接下来借助代码展开对内核启动部分的解释：



```c
asmlinkage void __init setup_vm(void)
{
        extern char _start;
        uintptr_t i;
        uintptr_t pa = (uintptr_t) &_start;
        pgprot_t prot = __pgprot(pgprot_val(PAGE_KERNEL) | _PAGE_EXEC);

        va_pa_offset = PAGE_OFFSET - pa;
        pfn_base = PFN_DOWN(pa);

        /* Sanity check alignment and size */
        BUG_ON((PAGE_OFFSET % PGDIR_SIZE) != 0);
        BUG_ON((pa % (PAGE_SIZE * PTRS_PER_PTE)) != 0);

#ifndef __PAGETABLE_PMD_FOLDED
        trampoline_pg_dir[(PAGE_OFFSET >> PGDIR_SHIFT) % PTRS_PER_PGD] =
                pfn_pgd(PFN_DOWN((uintptr_t)trampoline_pmd),
                        __pgprot(_PAGE_TABLE));
        trampoline_pmd[0] = pfn_pmd(PFN_DOWN(pa), prot);

        for (i = 0; i < (-PAGE_OFFSET)/PGDIR_SIZE; ++i) {
                size_t o = (PAGE_OFFSET >> PGDIR_SHIFT) % PTRS_PER_PGD + i;

                swapper_pg_dir[o] =
                        pfn_pgd(PFN_DOWN((uintptr_t)swapper_pmd) + i,
                                __pgprot(_PAGE_TABLE));
        }
        for (i = 0; i < ARRAY_SIZE(swapper_pmd); i++)
                swapper_pmd[i] = pfn_pmd(PFN_DOWN(pa + i * PMD_SIZE), prot);

        swapper_pg_dir[(FIXADDR_START >> PGDIR_SHIFT) % PTRS_PER_PGD] =
                pfn_pgd(PFN_DOWN((uintptr_t)fixmap_pmd),
                                __pgprot(_PAGE_TABLE));
        fixmap_pmd[(FIXADDR_START >> PMD_SHIFT) % PTRS_PER_PMD] =
                pfn_pmd(PFN_DOWN((uintptr_t)fixmap_pte),
                                __pgprot(_PAGE_TABLE));
#else
        trampoline_pg_dir[(PAGE_OFFSET >> PGDIR_SHIFT) % PTRS_PER_PGD] =
                pfn_pgd(PFN_DOWN(pa), prot);

        for (i = 0; i < (-PAGE_OFFSET)/PGDIR_SIZE; ++i) {
                size_t o = (PAGE_OFFSET >> PGDIR_SHIFT) % PTRS_PER_PGD + i;

                swapper_pg_dir[o] =
                        pfn_pgd(PFN_DOWN(pa + i * PGDIR_SIZE), prot);
        }

        swapper_pg_dir[(FIXADDR_START >> PGDIR_SHIFT) % PTRS_PER_PGD] =
                pfn_pgd(PFN_DOWN((uintptr_t)fixmap_pte),
                                __pgprot(_PAGE_TABLE));
#endif
}
```

这是初始化映射页表的函数，两个框的作用类似，只是上面的选择了四级页表设置，下面选择了三级页表设置。最终作用是生成初始一级线性映射的页表和三级页表的最大页表

```
    ##跳转到函数setup_vm建立临时页表
    la sp, init_thread_union + THREAD_SIZE
    call setup_vm
    ##利用relocate重定向，开启MMU并设置好satp
    call relocate

    ##设置init_task的CPU字段为零，设置初始任务的堆栈指针，正确地初始化任务的上下文和堆栈
    la tp, init_task
    sw zero, TASK_TI_CPU(tp)
    la sp, init_thread_union + THREAD_SIZE

    ##解析设备树二进制文件（DTB），并将解析结果存储在相应的数据结构中，开始执行内核的初始化代码
    mv a0, s1
    call parse_dtb
    tail start_kernel

relocate:
    ##计算出第一次映射的虚拟地址和物理地址相差的值至a1，即我们常用的PA2VA_OFFSET，将ra地址转换为虚拟地址以便MMU解析返回上层代码
    li a1, PAGE_OFFSET
    la a0, _start
    sub a1, a1, a0
    add ra, ra, a1

    ##以下的中断表述可能不太准确，应该叫异常或直接叫trap（
    ##设置中断程序的入口地址为虚拟地址，这个值我们将用作稍后的缺页异常入口来解决无等值映射的问题，这里的作用需要记住原处理trap的入口地址从物理地址转化为虚拟地址。
    ##当存在物理地址对应的指令想要执行时，此时没有等值映射，我们无法通过MMU找到对应的物理地址，此时产生缺页fault，跳入mtvec对应中断程序，由于我们刚刚此处的修改，这个地址是trap_handler的虚拟地址，是可以正常执行的，我们可以大胆猜测一下，他将此时的sepc利用偏移量计算出sepc对应的虚拟地址，中断处理结束后返回即可以通过MMU正常找到中断对应的物理地址
    la a0, 1f
    add a0, a0, a1
    csrw CSR_STVEC, a0

    ##计算出三级页表的最大页表地址后，将其转化为satp格式。最高几位利用SATP_MODE宏存储到a1寄存器，swapper_pg_dir经过页表大小右移后作为satp的PPN部分，最后利用or指令将预算的satp存储到a2，但是此时还暂不设置为satp
    la a2, swapper_pg_dir
    srl a2, a2, PAGE_SHIFT
    li a1, SATP_MODE
    or a2, a2, a1

    ##加载trampoline_pg_dir这个线性映射一级页表的地址到satp，和上文三级页表设置一样，最高几位利用SATP_MODE宏存储到a1寄存器，trampoline_pg_dir经过页表大小右移后作为satp的PPN部分，最后利用or指令将预算的satp存储到a2，并使用csrw指令修改satp开启MMU，至此我们开启了虚拟地址访存模式。
    la a0, trampoline_pg_dir
    srl a0, a0, PAGE_SHIFT
    or a0, a0, a1
    sfence.vma
    csrw CSR_SATP, a0
.align 2
1:
    ##设置trap入口,这里仅作debug用，一般不会跳入
    la a0, .Lsecondary_park
    csrw CSR_STVEC, a0

    ##加载全局指针
.option push
.option norelax
    la gp, __global_pointer$
.option pop

    ## 从中断处理回来之后跳入这里：设置swapper_pg_dir这个三级页表最高级的虚拟地址到satp，并使用csrw指令修改satp开启MMU，再次开启虚拟地址访存模式，此时是正常的三级页表
    csrw CSR_SATP, a2
    sfence.vma
    ##至此三级页表的重定向完成
    ret
```

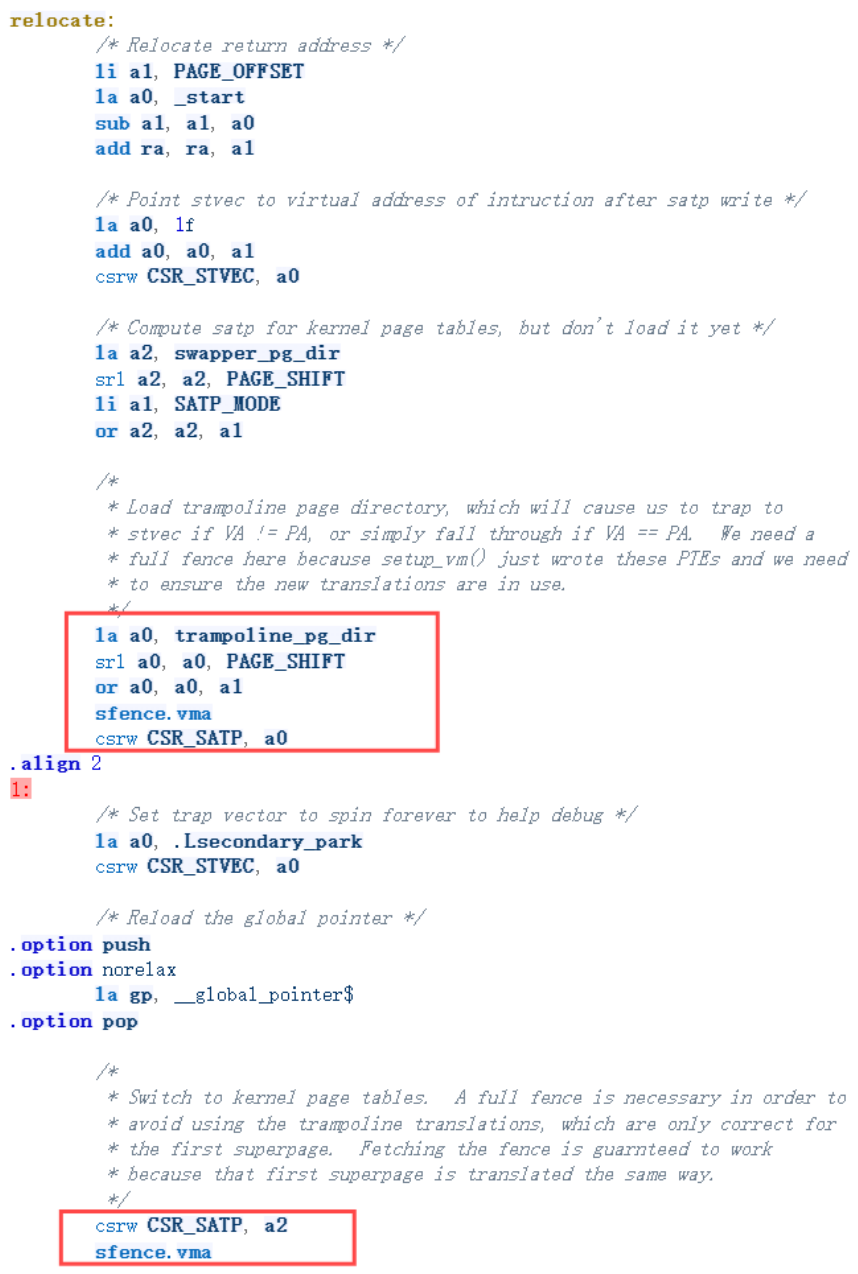
- 回答 Linux 为什么可以不进行等值映射，它是如何在无等值映射的情况下让 pc 从物理地址跳到虚拟地址；

  Linux使用`trap`处理程序处理了在等值映射产生的异常，具体原理如下：

1. `_start`中，在设置`satp`之前设置中断程序的入口地址`mtvec`为虚拟地址，这个值我们将用作稍后的缺页异常入口，将原处理trap的入口地址从物理地址转化为虚拟地址。
2. 当存在物理地址对应的指令想要执行时，此时没有等值映射，我们无法通过MMU找到对应的物理地址，此时产生缺页fault，跳入`mtvec`对应中断程序，由于我们刚刚此处的修改，这个地址是`trap_handler`的虚拟地址，是可以正常执行的，将此时的`sepc`利用偏移量计算出`sepc`对应的虚拟地址，中断处理结束后返回即可以通过MMU正常找到**中断指令对应的物理地址**。

- 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` 通常是在启动过程中设置的临时页表，是线性映射，保证了初始化代码的正常执行；另外由于其映射线性的易操作性，用作初始化内核的debug用。
  - `swapper_pg_dir` 是内核的主页表目录，用于在正常运行时管理内存映射，sv39模式下有三级页表，符合层级映射。

在`relocate`的两个`csrw`处设置：



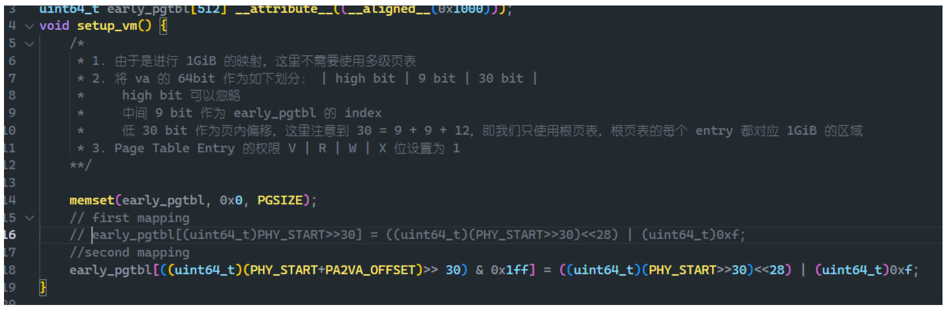
```
relocate:
        /* Relocate return address */
        li a1, PAGE_OFFSET
        la a0, _start
        sub a1, a1, a0
        add ra, ra, a1

        /* Point stvec to virtual address of intruction after satp write */
        la a0, 1f
        add a0, a0, a1
        csrw CSR_STVEC, a0

        /* Compute satp for kernel page tables, but don't load it yet */
        la a2, swapper_pg_dir
        srl a2, a2, PAGE_SHIFT
        li a1, SATP_MODE
        or a2, a2, a1

        /*
         * Load trampoline page directory, which will cause us to trap to
         * stvec if VA != PA, or simply fall through if VA == PA.  We need a
         * full fence here because setup_vm() just wrote these PTEs and we need
         * to ensure the new translations are in use.
         */
        la a0, trampoline_pg_dir
        srl a0, a0, PAGE_SHIFT
        or a0, a0, a1
        sfence.vma
        csrw CSR_SATP, a0
.align 2
1:
        /* Set trap vector to spin forever to help debug */
        la a0, .Lsecondary_park
        csrw CSR_STVEC, a0

        /* Reload the global pointer */
.option push
.option norelax
        la gp, __global_pointer$
.option pop

        /*
         * Switch to kernel page tables.  A full fence is necessary in order to
         * avoid using the trampoline translations, which are only correct for
         * the first superpage.  Fetching the fence is guarnteed to work
         * because that first superpage is translated the same way.
         */
        csrw CSR_SATP, a2
        sfence.vma
```

第一个红框是 `trampoline_pg_dir` 被设置为页表

第二个红框是 `swapper_pg_dir` 被设置为页表

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

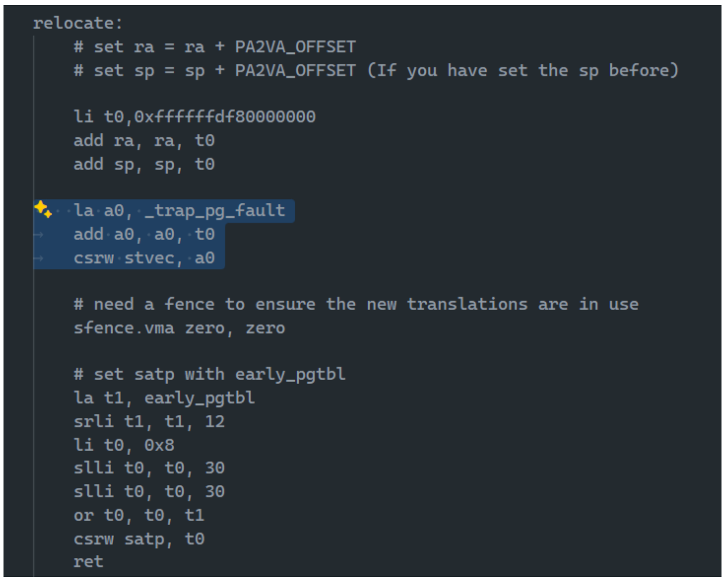
  1. 在 `vm.c` 中注释掉等值映射：



```c
uint64_t early_pgtbl[512] __attribute__((__aligned__(0x1000)));
void setup_vm() {
    /*
     * 1. 由于是进行 1GiB 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
     *     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 1GiB 的区域
     * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    **/

    memset(early_pgtbl, 0x0, PGSIZE);
    // first mapping
    // early_pgtbl[(uint64_t)PHY_START>>30] = ((uint64_t)(PHY_START>>30)<<28) | (uint64_t)0xf;
    //second mapping
    early_pgtbl[((uint64_t)(PHY_START+PA2VA_OFFSET)>> 30) & 0x1ff] = ((uint64_t)(PHY_START>>30)<<28) | (uint64_t)0xf;
}
```

2. 在`head.S`的`relocate`添加对**trap**处理入口的虚拟地址化

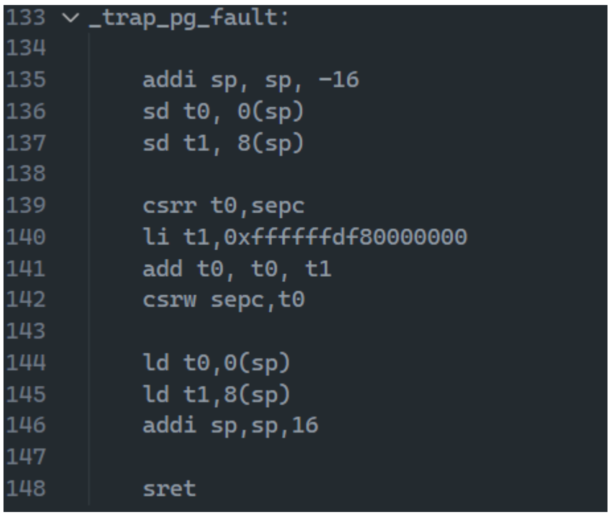
> 注意extern引入_trap_pg_fault



```
relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)

    li t0,0xffffffdf80000000
    add ra, ra, t0
    add sp, sp, t0

    la a0, _trap_pg_fault
    add a0, a0, t0
    csrw stvec, a0

    # need a fence to ensure the new translations are in use
    sfence.vma zero, zero

    # set satp with early_pgtbl
    la t1, early_pgtbl
    srli t1, t1, 12
    li t0, 0x8
    slli t0, t0, 30
    slli t0, t0, 30
    or t0, t0, t1
    csrw satp, t0
    ret
```

3. 此时注意到我们新建了**trap**处理程序`_trap_pg_fault`，该汇编程序我们可以放置在`entry.S`中

> 注意.global显式声明_trap_pg_fault供外部调用

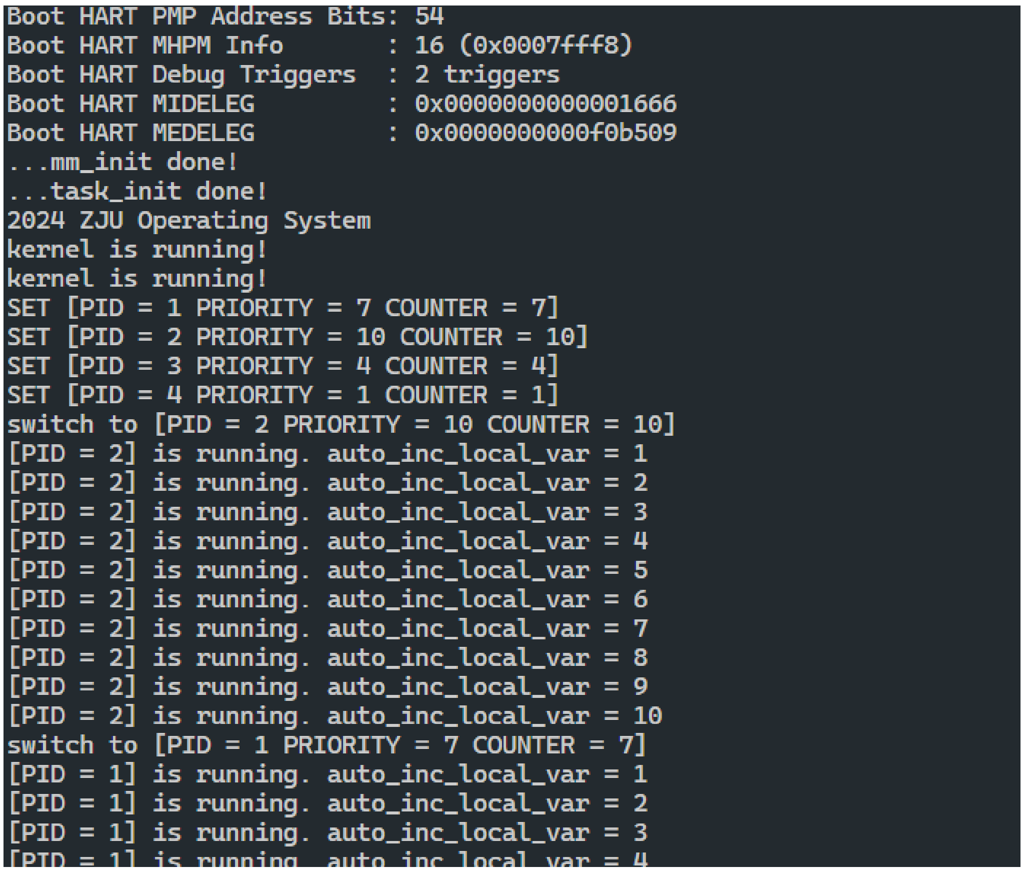


```
_trap_pg_fault:

    addi sp, sp, -16
    sd t0, 0(sp)
    sd t1, 8(sp)

    csrr t0,sepc
    li t1,0xffffffdf80000000
    add t0, t0, t1
    csrw sepc,t0

    ld t0,0(sp)
    ld t1,8(sp)
    addi sp,sp,16

    sret
```

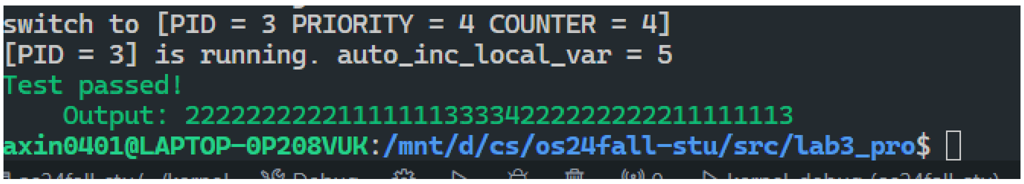
这里我们保存了上下文，更改了返回地址为虚拟地址

之后我们就可以愉快地运行了，运行代码一并交至学在浙大：

> make TEST_SCHED=1 run运行

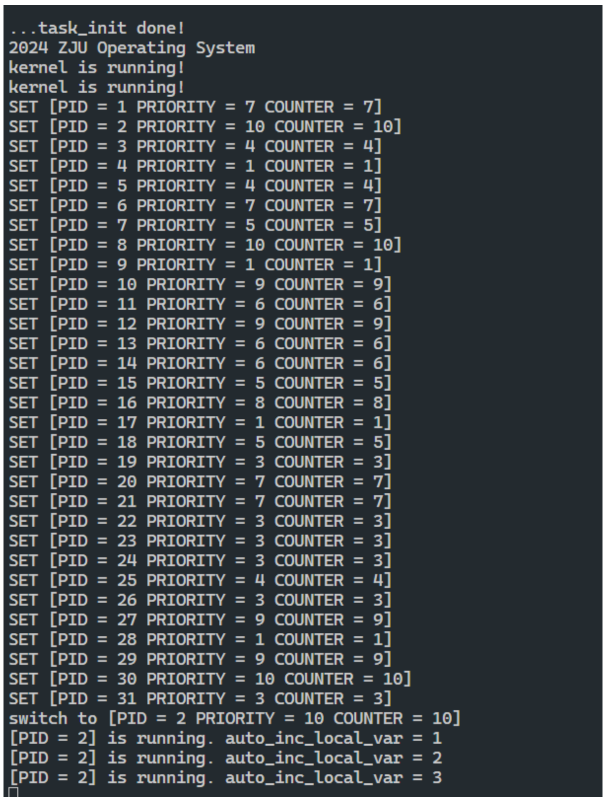


```
Boot HART PMP Address Bits: 54
Boot HART MHPM Info         : 16 (0x0007fff8)
Boot HART Debug Triggers    : 2 triggers
Boot HART MIDELEG           : 0x0000000000001666
Boot HART MEDELEG           : 0x0000000000f0b509
...mm_init done!
...task_init done!
2024 ZJU Operating System
kernel is running!
kernel is running!
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 1
[PID = 2] is running. auto_inc_local_var = 2
[PID = 2] is running. auto_inc_local_var = 3
[PID = 2] is running. auto_inc_local_var = 4
[PID = 2] is running. auto_inc_local_var = 5
[PID = 2] is running. auto_inc_local_var = 6
[PID = 2] is running. auto_inc_local_var = 7
[PID = 2] is running. auto_inc_local_var = 8
[PID = 2] is running. auto_inc_local_var = 9
[PID = 2] is running. auto_inc_local_var = 10
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[PID = 1] is running. auto_inc_local_var = 1
[PID = 1] is running. auto_inc_local_var = 2
[PID = 1] is running. auto_inc_local_var = 3
[PID = 1] is running. auto_inc_local_var = 4
```



```
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 5
Test passed!
    Output: 2222222222111111133334222222222211111113
axin0401@LAPTOP-0P208VUK:/mnt/d/cs/os24fall-stu/src/lab3_pro$
```

> make run 运行截图



```
...task_init done!
2024 ZJU Operating System
kernel is running!
kernel is running!
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]
SET [PID = 5 PRIORITY = 4 COUNTER = 4]
SET [PID = 6 PRIORITY = 7 COUNTER = 7]
SET [PID = 7 PRIORITY = 5 COUNTER = 5]
SET [PID = 8 PRIORITY = 10 COUNTER = 10]
SET [PID = 9 PRIORITY = 1 COUNTER = 1]
SET [PID = 10 PRIORITY = 9 COUNTER = 9]
SET [PID = 11 PRIORITY = 6 COUNTER = 6]
SET [PID = 12 PRIORITY = 9 COUNTER = 9]
SET [PID = 13 PRIORITY = 6 COUNTER = 6]
SET [PID = 14 PRIORITY = 6 COUNTER = 6]
SET [PID = 15 PRIORITY = 5 COUNTER = 5]
SET [PID = 16 PRIORITY = 8 COUNTER = 8]
SET [PID = 17 PRIORITY = 1 COUNTER = 1]
SET [PID = 18 PRIORITY = 5 COUNTER = 5]
SET [PID = 19 PRIORITY = 3 COUNTER = 3]
SET [PID = 20 PRIORITY = 7 COUNTER = 7]
SET [PID = 21 PRIORITY = 7 COUNTER = 7]
SET [PID = 22 PRIORITY = 3 COUNTER = 3]
SET [PID = 23 PRIORITY = 3 COUNTER = 3]
SET [PID = 24 PRIORITY = 3 COUNTER = 3]
SET [PID = 25 PRIORITY = 4 COUNTER = 4]
SET [PID = 26 PRIORITY = 3 COUNTER = 3]
SET [PID = 27 PRIORITY = 9 COUNTER = 9]
SET [PID = 28 PRIORITY = 1 COUNTER = 1]
SET [PID = 29 PRIORITY = 9 COUNTER = 9]
SET [PID = 30 PRIORITY = 10 COUNTER = 10]
SET [PID = 31 PRIORITY = 3 COUNTER = 3]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 1
[PID = 2] is running. auto_inc_local_var = 2
[PID = 2] is running. auto_inc_local_var = 3
```