浙江大学

本科实验报告

课程名称：操作系统

姓　　名：徐文皓

学　　院：计算机科学与技术学院

系：软件工程系

专　　业：软件工程

学　　号：3210102377

指导教师：夏莹杰

2023 年　　11 月　　20 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：____手机：____

实验地点：线上 实验日期：2023 年 11 月 20 日

一、实验目的和要求

本实验的目的有：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。

本实验的要求是：独立完成作业；在 lab2 的基础上，修改 defs.h 文件；并且从仓库拉取 vmlinux.lds 和 Makefile 文件；进行虚拟内存映射，实现 setup_vm 和 setup_vm_final 函数。

二、实验过程

在 defs.h 中追加指定内容。

```
#define OPENSBI_SIZE (0x200000)

#define VM_START (0xfffffffe00000000)
#define VM_END   (0xffffffff00000000)
#define VM_SIZE  (VM_END - VM_START)

#define PA2VA_OFFSET (VM_START - PHY_START)
```

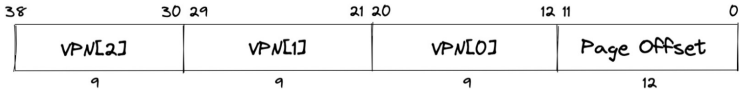
将本实验新增的文件添加到相应位置。在顶层的 Makefile 中做出如下更改，使得内核可以使用刷新缓存的指令扩展，并自动在编译项目前执行 clean 任务来防止对头文件的修改无法触发编译任务。

```
ISA=rv64imafd_zifencei
...
all: clean
...
```

在 arch/riscv/kernel 目录下新建 vm.c 文件，实现 setup_vm()。pa 和 va 分别代表物理地址和虚拟地址，mid_bits 是用于取 Pagetable Entry[38:30]的掩码，low_bits 是用于取 Pagetable Entry[29:0]的掩码。

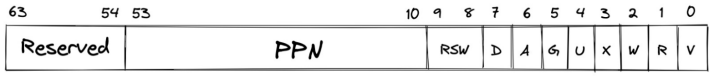
```c
// arch/riscv/kernel/vm.c
# include "defs.h"
/* early_pgtbl: 用于 setup_vm 进行 1GB 的 映射。 */
unsigned long  early_pgtbl[512] __attribute__((__aligned__(0x1000)));

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 1GB 的区域。
    3. Page Table Entry 的权限 V | R | W | X 位设置为 1
    */
    unsigned long va = 0x80000000;
    unsigned long pa = 0x80000000;
    unsigned long mid_bits = 0x7fc0000000; // | *9bits* |  30bits  |
    unsigned long low_bits = 0x003fffffff; // |  9bits  | *30bits* |

    /* pa>>12得到物理页码
     * (pa>>12)<<10将物理页码置于page table entry的PPN[0]位置
     * ((pa>>12)<<10)|0xf将page table entry的XWRV共4位均设置为1
     * 不直接写成>>2，因为原写法可以清除低12位的OFFSET
     */
    early_pgtbl[(va & mid_bits) >> 30] = ((pa >> 12) << 10) | 0xf;

    // 将va转变至高地址
    va = va + PA2VA_OFFSET;
    early_pgtbl[(va & mid_bits) >> 30] = ((pa >> 12) << 10) | 0xf;
}
```

考虑 SV39 Virtual Address 的结构。其中 VPN[2]，即 va[38:30]，对应 early_pgtbl 的下标，因此只需要将 early_pgtbl[va[38:30]]这一 SV39 Pagetable Entry 设置为对应物理地址的映射即可。



SV39 Virtual Address

考虑 SV39 Pagetable Entry 的结构。Pagetable Entry[53:10]存放物理页码，即物理地址的[55:12]。Pagetable Entry[9:0]是 protection bits，在本次实验中我们将[9:4]设置为 0，[3:0](XWRV)根据实验需要设置。在 setup_virtual 中，我们将 XWRV 均设置为 1。因此，形如 early_pgtbl[(va&mid_bits)>>30]=((pa>>12)<<10) | 0xf 的指令完成了一次从物理地址到虚拟地址的映射。



SV39 Pagetable Entry

首先，将 va 和 pa 均设置为 0x80000000，使用如上代码进行等值映射。随后，通过 va=va+PA2VA_OFFSET 将虚拟地址加载到高地址，再次使用如上代码完成一次映射。

由于 vmlinux.lds 被更改，在 head.S 中，经过 gdb 调试，我们发现此时 sp 已经被设置为虚拟地址。但由于此时 setup_vm()等尚未被调用，因此我们先通过 sp=sp-PA2VA_OFFSET 将其设置回物理地址，对 setup_vm()进行调用。返回后，物理地址到虚拟地址的第一次映射已完成，调用 relocate。

```
_start:
    la sp, boot_stack_top
    li t0, 0xffffffdf80000000 // PA2VA_OFFSET
    sub sp, sp, t0

    call setup_vm
    call relocate
    call mm_init
```

在 relocate 中，将 ra 和 sp 移动到虚拟地址，并对 satp 进行设置。

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

    li t0, 0xffffffdf80000000 // PA2VA_OFFSET
    add ra, ra, t0
    add sp, sp, t0

    # set satp with early_pgtbl
    la t0, early_pgtbl
    li t1, 0xffffffdf80000000 // PA2VA_OFFSET
    sub t0, t0, t1
    srli t0, t0, 12
    li t1, 0x8000000000000000
    or t0, t0, t1
    csrw satp, t0

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret
```

考虑 satp 的结构。satp[43:0](PPN)存放顶级页表(即 early_pgtbl)对应的物理页号，即(early_pgtbl-PA2VA_OFFSET)>>12。satp[59:44](ASID)在本次实验中置零。satp[63:60](MODE)设置为 8，即选择 SV39 模式。

satp Register

从 recolate 返回后，gdb 显示跳转到高地址处继续执行。稍后在 set_vm_final()中，我们完成对其余地址的映射。

在进行这一步之前，现在 mm.c 中修改 mm_init 函数，实现对内存的初始化。考虑 kfreerange 的函数原型为 kfreerange(char *start, char *end)，而我们观察到，在 vmlinux.lds 中，第一个实际参数_skernel 的地址已经被修改到高地址，本次实验中由物理地址到虚拟地址的映射为线性映射，因此只需要将第二个参数修改为

(char *)PHY_END+PA2VA_OFFSET 即可。

```
/* kernel代码起始位置 */
BASE_ADDR = VM_START + OPENSBI_SIZE;
...
 /* . 代表当前地址 */
. = BASE_ADDR;
/* 记录kernel代码的起始地址 */
_skernel = .;
...
```

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

在 vmlinux.lds 中，可以观察到_stext 和_etext 分别代表 text 段的起始和结束地址，以此类推。而_sdata 代表其余部分的起始地址。因此，我们将这些地址标识符声明到 vm.c 中，以便 setup_vm_final()使用。

```
/* _stext, _etext 分别记录了text段的起始与结束地址 */
    .text : ALIGN(0x1000){
        _stext = .;
        ...
        _etext = .;
    } >ramv AT>ram

    .rodata : ALIGN(0x1000){
        _srodata = .;
        ...
        _erodata = .;
    } >ramv AT>ram

    .data : ALIGN(0x1000){
        _sdata = .;
    ...
```

在 setup_vm_final 中，分别调用 create_mapping，对 text、rodata 和其余部分分别建立映射，起始地址和结束地址用刚刚引入的外部声明表示，长度为两者之差，并通过最后一个参数设置页权限。需要特别指明的是，对于其余部分，长度为 PHY_END+PA2VA_OFFSET-(uint64)_sdata。在完成映射后，将 satp 的 PPN 部分设置为 swapper_pg_dir 所对应的物理地址。

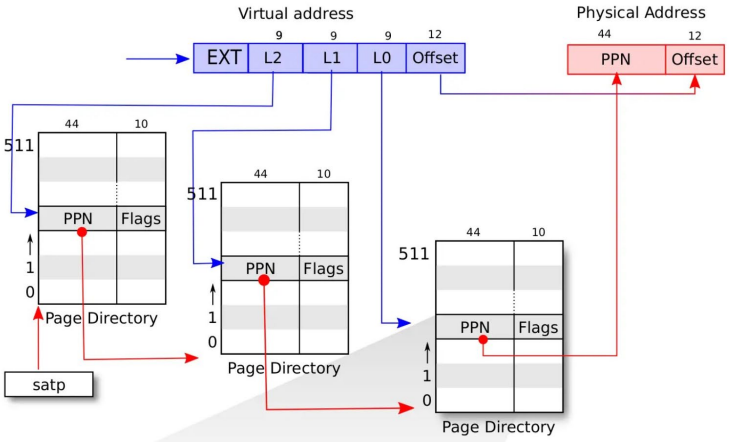
```
unsigned long  swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
extern char _stext[], _etext[];
extern char _srodata[], _erodata[];
extern char _sdata[], _edata[];

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

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

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

    // mapping other memory -|W|R|V
    create_mapping(swapper_pg_dir,_sdata,_sdata-PA2VA_OFFSET,PHY_END+PA2VA_OFFSET-(uint64)_sdata,0x7);

    // set satp with swapper_pg_dir

    register unsigned long t0 asm("t0") = (unsigned long)swapper_pg_dir - PA2VA_OFFSET;
    t0 = (t0 >> 12) | 0x8000000000000000;
    asm volatile("csrw satp, t0" : : "r"(t0) : "memory");

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

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

接下来考虑 create_mapping()。为便于操作，我们引入 create_mapping_page()，并对 create_mapping()按页进行映射。由于 va 和 pa 均为地址而非页码，因此步长需要设置为 PGSIZE。

```c
create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm) {
    for (int i = 0; i < sz; i = i + PGSIZE)
        create_mapping_page(pgtbl, va + i, pa + i, perm);
}
```

在 create_mapping_page()中，分别使用掩码取 va 的不同位，获得 3 个 VPN。首先查看 pgtbl[VPN[2]]，它对应第一级页表的 pte，通过 0x1 掩码得到 valid 位，如果为 0，说明该页表项还不存在，我们通过 kalloc()获得一页作为页表目录，以其物理地址和 0x1 的 XWRV 设置 Pagetable Entry，作为 pgtbl[VPN[2]]。随后，通过 pgtbl = (unsigned long *)((pgtbl[VPN[2]] >> 10 << 12) + PA2VA_OFFSET)来到这个页表，对下一级页表进行类似操作。在第三级页表时，它将指向物理地址而非下一个页表，因此我们将这个 Pagetable Entry 的 XWRV 设置为指定参数，当 XWR 不为 0 时说明该 Pagetable Entry 所指向的是物理地址。

```c
create_mapping_page(uint64 *pgtbl, uint64 va, uint64 pa, int perm){
    unsigned long VPN[3];
    VPN[2] = (va >> 30) & 0x1ff;
    VPN[1] = (va >> 21) & 0x1ff;
    VPN[0] = (va >> 12) & 0x1ff;

    unsigned long pte = pgtbl[VPN[2]]; // 第一级映射
    if ((pte & 0x1) == 0) { // 如果不存在页表
        unsigned long pgtbl_tmp = (unsigned long)kalloc() - PA2VA_OFFSET;
        pgtbl_tmp = (pgtbl_tmp >> 12) << 10;
        pgtbl[VPN[2]] = (unsigned long)pgtbl_tmp | 0x1;
    }

    pgtbl = (unsigned long *)((pgtbl[VPN[2]] >> 10 << 12) + PA2VA_OFFSET);

    pte = pgtbl[VPN[1]]; // 第二级映射
    if ((pte & 0x1) == 0) { // 如果不存在页表
        unsigned long pgtbl_tmp = (unsigned long)kalloc() - PA2VA_OFFSET;
        pgtbl_tmp = (pgtbl_tmp >> 12) << 10;
        pgtbl[VPN[1]] = (unsigned long)pgtbl_tmp | 0x1;
    }

    pgtbl = (unsigned long *)((pgtbl[VPN[1]] >> 10 << 12) + PA2VA_OFFSET);

    pte = ((pa >> 12) << 10) | perm;
    pgtbl[VPN[0]] = pte;
}
```



至此，我们完成了 setup_vm_final()的相关设置，在 head.S 中将其置于 mm_init 和 task_init 之间即可。

```
...
call mm_init
call setup_vm_final
call task_init
...
```

至此，本实验的全部任务已经完成。

为便于观察实验现象，我们修改 arch/riscv/kernel/proc.c，在 dummy()函数中将原来的输出信息追加上进程地址。

```
printk("[PID = %d] is running. thread space begins at 0x%llx\n", current->pid, current);
```

启动内核，可以观察到以下现象。

```
...proc_init done!
2022 Hello RISC-V
[S] Supervisor Mode Timer Interrupt

SWITCH TO [pid=8 counter=1 priority=25]
[PID = 8] is running. thread space begins at 0xffffffe007fb7000
[S] Supervisor Mode Timer Interrupt

SWITCH TO [pid=7 counter=2 priority=5]
[PID = 7] is running. thread space begins at 0xffffffe007fb8000
[S] Supervisor Mode Timer Interrupt
[PID = 7] is running. thread space begins at 0xffffffe007fb8000
[S] Supervisor Mode Timer Interrupt

SWITCH TO [pid=14 counter=2 priority=6]
[PID = 14] is running. thread space begins at 0xffffffe007fb1000
[S] Supervisor Mode Timer Interrupt
[PID = 14] is running. thread space begins at 0xffffffe007fb1000
[S] Supervisor Mode Timer Interrupt
```

三、讨论和心得

本次实验理解起来较有难度，尤其是对内存的操作使得 gdb 并不能像前几次实验那样正常工作，但是在同学的热心帮助下实验得以较为顺利地完成。实验中，可以通过 gdb 输出相应变量、函数的地址，检查其位置是否处于虚拟地址。过程中遇到了以下问题：

在外部声明_stext 等值时，我起初使用 char*类型，无法正常运行，经过 gdb 调试，发现&_stext 才是其地址。将声明改为 char[]类型或者将 create_mapping() 相应实际参数更改为形如&_stext 的格式，问题得以解决。这里启示我们在对值进行强制类型转换时，转换到指针类型和转换到数组类型的区别。

建议在实验指导书中对 vmlinux.lds 的更改做一些说明。

四、思考题

1. 验证.text, .rodata 段的属性是否成功设置，给出截图。

即验证.text 段的 XWR 是否为 101，验证.rodata 段的 XWR 是否为 001。

程序可以正常执行，说明.text 段的 X 位成功设置为 1。

要验证某个段是否具有某种权限，只需要验证在对该段进行某种操作时是否会抛出异常即可。在之前的实验中，我们已经在中断处理函数中对非时钟中断输出 trap 原因，可以借此来进行观察。

```
//arch/riscv/kernel/trap.c
...
if(interrupt&&cause==5){ // timer interrupt
      printk("[S] Supervisor Mode Timer Interrupt\n");
      clock_set_next_event();
      do_timer();
  }
  else{
      printk("[Error] Not Timer Interrupt with scause = 0x%16x\n", scause);
  }
...
```

然而，非时钟中断会导致程序一直返回同一行进而出现死循环。为了打破这个局面，我们在处理其他中断时对 sepc 加 4。考虑到 entry.S 中是由_traps 在调用 trap_handler()前后负责维护 sepc 的稳定性，因此在 trap_handler()中实现这一操作并不可行。

所以，我们在_traps 中实现这一功能：在从 trap_handler 返回后，如果 scause 表明本次中断是时钟中断则将 sepc 恢复，否则将 sepc 恢复后加 4。

```
# arch/riscv/kernel/entry.S
...
    call trap_handler

    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
    ld a0, 256(sp)
    li t0, 0x8000000000000005 # temporarily add 4 to sepc
    csrr t1, scause
    beq t1, t0, _csrwrite
    addi a0, a0, 4
_csrwrite:
    csrw sepc, a0
...
```

至此，我们可以通过观察异常信息检查两个段的权限。

首先检验.rodata 段的 X 权限。在 head.S 中，我们完成中断设置后，执行 j _srodata 指令，即试图跳转到.rodata 段进行执行。

```
# arch/riscv/kernel/head.S
    csrs sstatus, 2 # set sstatus[SIE] = 1

    j _srodata
    call start_kernel
```

启动内核，发现给出了 scause 为 0xc 的异常提示。

```
...proc_init done!
[Error] Not Timer Interrupt with scause = 0x0000000c
```

对照 scause 的表，我们发现 0xc 为 Instruction Page Fault，说明内核无法从 _srodata 处读取指令，即不能执行.rodata 的内容。

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved for future standard use*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved for future standard use*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10–15	*Reserved for future standard use*
1	≥16	*Reserved 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 for future standard use*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

接下来验证 W 和 R 两个属性。

在 start_kernel()中，我们尝试打印_stext 的值，并尝试对其写入数据。

```c
// init/main.c
...
extern char _stext[];

int start_kernel() {
    ...

    printk("_stext = %x\n", *_stext);
    *_stext = 0;
    printk("_stext = %x\n", *_stext);

    ...
}
```

启动内核，观察到如下输出。能够正常打印其值，证明它是可读的。scause 为 0xf，对应 Store/AMO Page Fault，说明写操作被拒绝。相应地，在下一次打印中我们发现其值并没有发生变化，这同样说明了写操作失败。

```
...proc_init done!
2022 Hello RISC-V
_stext = 00000017
[Error] Not Timer Interrupt with scause = 0x0000000f
_stext = 00000017
[S] Supervisor Mode Timer Interrupt
```

类似地，对_srodata 进行验证，现象相同。

```
...proc_init done!
2022 Hello RISC-V
_srodata = 0000002e
[Error] Not Timer Interrupt with scause = 0x0000000f
_srodata = 0000002e
[S] Supervisor Mode Timer Interrupt
```

至此，我们验证了.text 和.rodata 段 XWR 属性的正确性。

2. 为什么我们在 setup_vm()中需要做等值映射?

在回答这个问题前，不妨将等值映射删去，观察现象。

```
// arch/riscv/kernel/vm.c

// early_pgtbl[(va & mid_bits) >> 30] = ((pa >> 12) << 10) | 0xf;
```

启动内核，意料之中地，无法正常运行。经过 gdb 调试，我们发现在 csrw satp,t0 时程序无法继续执行。这里的功能是将 satp 的 PPN 位设置为 early_pgtbl 的物理地址。

```
   0x8020009c      addiw   t1,zero,-1
   0x802000a0      slli    t1,t1,0x3f
>  0x802000a4      or      t0,t0,t1
   0x802000a8      csrw    satp,t0
   0x802000ac      sfence.vma
   0x802000b0      fence.i
   0x802000b4      ret
```

在调用 setup_vm_final()完成对所有页表项的设置之前，我们的内核还在*0x8020000 处运行。在修改 satp 后、ret 执行前程序仍在物理地址处运行，如果在此时不进行等值映射，随后的命令(在本例中，对应*0x802000ac 的 sfence.vma 指令，它已经认为这个地址是虚拟地址而非物理地址)将无法从新的 satp 所指向的页表中查找到实际的物理地址，进而无法执行。

为了验证这里是由于命令无法执行所导致程序无法正常运行，我们在等值映射时通过修改参数改变该页的权限。当参数为 0xf 时，即实验中的情况，我们已经知道可以正常运行。将参数修改为 0x7，即授予 WR 权限而不授予 X 权限，仍未在 csrw satp,t0 命令处无法继续执行，问题并没有解决。将参数修改为 0x9，即授予 X 权限而不授予 WR 权限，可以发现程序可以正常运行。

因此，我们判断是 satp 发生变更后的三条指令仍需要访问页表确定自己的物理地址，进而被执行。而等值映射为这三条指令能够“找到”自己的物理地址提供了保证。

3. 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。

经由上问中我们所发现的程序正常运行与该页 X 权限的依赖关系这一事实，我们可以发现在 satp 发生变更后，程序在执行下一条指令 sfence.vma 时，会触发 Instruction Page Fault 这一异常。考虑 trap 处理函数，这将是我们的突破口。

然而，如果能够正常进入 trap 处理函数，根据我们 trap_handler()的设计逻辑，我们所观察到的现象应该是时钟中断和其他 trap 中的某一种输出。程序呈现卡死状态，说明这一函数并没有顺利执行。经过思考，我们发现问题在于 stvec 受到了虚拟地址的影响，无法从新的 satp 所指向的页表中查找到实际的物理地址。

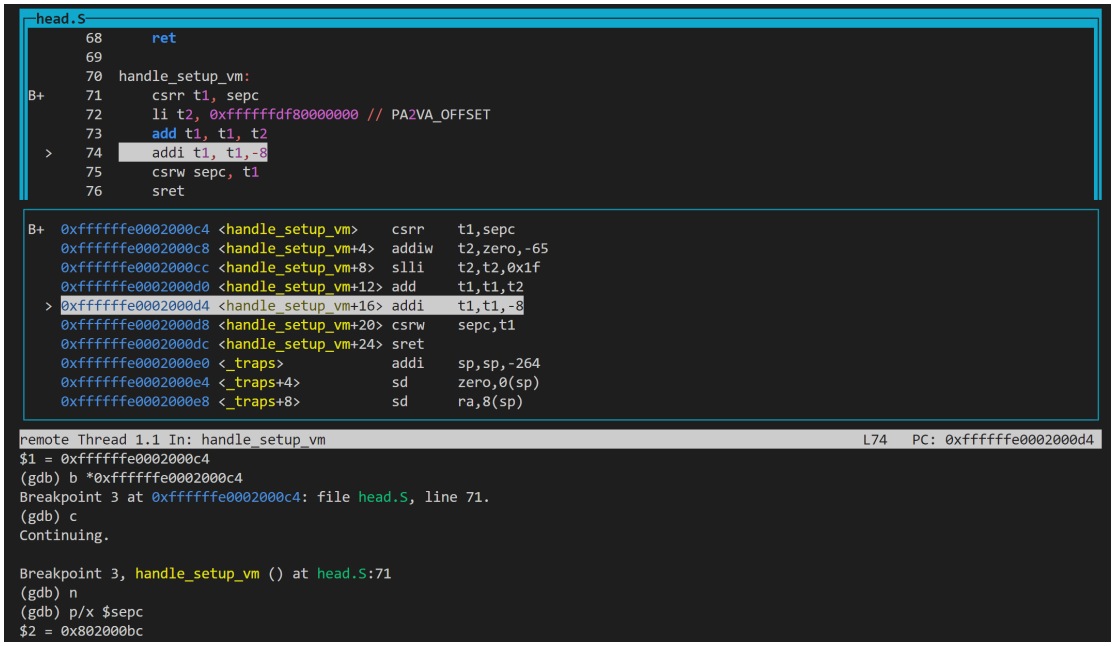
为此，不妨考虑更新 stvec 的值，使其能够被正常访问。在 head.S 的 relocate 中，设置 satp 前，我们临时修改 stvec 的值，使得 satp 设置后程序产生异常能够跳转到 handle_setup_vm，在这里我们讲 sepc 调整至正确地址。

```
relocate:
    ...
    la t1,handle_setup_vm
    csrw stvec, t1

    csrw satp, t0

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret

handle_setup_vm:
    csrr t1, sepc
    li t2, 0xffffffdf80000000 // PA2VA_OFFSET
    add t1, t1, t2
    addi t1, t1,-4
    csrw sepc, t1
    sret
```

需要特别说明的是，异常与中断不同，发生异常时 sepc 的值会被设置为异常指令的下一条指令地址，但在本例中我们需要发生异常的 sfence.vma 语句完成执行，因此，在设置 sepc 时，我们引入了-4 这一偏移，即向前一条指令。

下面给出实验验证：

在执行到临时修改 stvec 的位置，我们发现 sepc 的值为 0x802000bc。



而我们预期在异常返回后的地址为 0x802000b8。

因此，-4 偏移是合理的。

这样，内核即可在不作等值映射的条件下正常运行。