# 浙江大学


| | |
|---|---|
| 课程名称： | 操作系统原理与实践 |
| 题　　目： | Lab 3: RV64 虚拟内存管理 |
| 授课教师： | 申文博 |
| 助　　教： | 王鹤翔、陈淦豪、许昊瑞 |
| 姓　　名： | 潘潇然 |
| 学　　号： | 3220106049 |
| 地　　点： | 32舍367 |

## 一、实验过程与步骤

1. 准备工程
   - 在 `defs.h` 添加以下内容，作为我们在本次实验中需要用到的宏

```
#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)
```

   - 关闭PIE，保证实验正确性。在Makefile的 `CF` 中加入 `-fno-pie`
2. `setup_vm` 的实现
   - 首先我们需要将 `0x80000000` 开始的1GB区域进行两次映射，其中先进行一次等值映射，再进行一次加上 `PV2VA_OFFSET` 作为偏移量的映射，使其映射到 `direct mapping area`

```
/* 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);
  // 设置 PA == VA 的等值映射
  // [53:28] = [55:30]
  uint64_t index = ((uint64_t)(PHY_START) >> 30) & 0x1ff;  // 取[38:30]
  early_pgtbl[index] = (((PHY_START >> 30) & 0x3ffffff) << 28) | 0xf;

  // 设置 PA + PA2VA_OFFSET == VA 的映射
  index = ((uint64_t)(VM_START) >> 30) & 0x1ff;
  early_pgtbl[index] = (((PHY_START >> 30) & 0x3ffffff) << 28) | 0xf;
  printk("..setup_vm done!\n");
}
```

在SV39中，虚拟地址只有低39位有效，其中 `[38:30]` 表示 `VPN[2]` ，`[29,21]` 表示 `VPN[1]` ，`[20:12]` 表示 `VPN[0]` 。物理地址中 `[55:30]` 表示 `PPN[2]` ，`[29:21]` 表示 `PPN[1]` ，`[20:12]` 表示 `PPN[0]` 。在页表项中， `[53:28]` 表示 `PPN[2]` ，`[27:19]` 表示 `PPN[1]` ，`[18:10]` 表示 `PPN[0]`

因此在等值映射中，将 `PHY_START` 的 `[38,30]` 取出来作为页表项的index，之后取出其 `[55:30]` 位，并将最后四位分别表示V,R,W,X的置为1即可

在 `PA + PA2VA_OFFSET == VA` 的映射中，我们只需要将加上 `PA2VA_OFFSET` 的地址取出其 `[38:30]` 作为页表项的index即可，其他均保持一致。

- 之后我们需要通过 `relocate` 函数完成对 `satp` 的设置

```
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

    li t2, 8
    slli t2, t2, 60
    la t1, early_pgtbl
    srli t1, t1, 12
    or t1, t1, t2
    csrw satp, t1

    ret
```

这里我们首先通过伪指令 `li` 载入 `PA2VA_OFFSET` ，并将 `ra` 和 `sp` 分别加上此偏差。之后在设置satp前我们需要先运行 `sfence.vma` 来保证新的页表项生效。在本次实验中我们使用SV39，因此我们将 `[63:60]` 设置为8。然后我们将页表项加载进来，将其右移12位得到44位的PPN，将以上得到的两者取或即可得到satp的值。

- 之后我们在 `head.S` 的 `_start` 中的适当位置加入以上两个函数

```
_start:

    la sp, boot_stack_top
    jal setup_vm
    jal relocate

    jal mm_init
```

- 同时修改 `mm_init` 函数，将结束地址调整为虚拟地址

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

3. `setup_vm_final` 的实现：我们在调用 `mm_init` 完成内存管理初始化后，调用 `setup_vm_final` 需要完成对所有物理内存的映射，并设置正确的权限
   - `create_mapping` 函数
     - 首先利用 `va` 和 `sz` 计算得到映射范围，当虚拟地址还没到映射末尾时，我们在每次循环结束加 `PG_SIZE` 切换到下一页
     - 在每次循环，我们首先取出三级页表分别对应的 `vpn`
     - 首先对第一级页表，我们根据根页表基地址加上index得到对应的页表项。如果该页表项存在，即Valid Bit为1，我们就直接取出页表项，右移10位清零flag，之后左移12位到正确的位置，并记得需要加上 `PA2VA_OFFSET` 。若该页表不存在，则使用 `kalloc()` 获取一页，但注意到这里获取的是虚拟地址，需要我们减去 `PA2VA_OFFSET` 得到物理地址。之后我们对应地将其右移12位得到PPN，之后左移10位到正确的位置并加上 `PA2VA_OFFSET` ，最后将末位的Valid Bit置为1。对第二级页表，我们也采取类似的操作
     - 在第三级页表，我们直接将pte设置为pa的PPN并设置权限和Valid Bit即可

```
void create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz,
                    uint64_t perm) {
  uint64_t va_end = va + sz;
  uint64_t vpn2, vpn1, vpn0;
  while (va < va_end) {
    vpn2 = (va >> 30) & 0x1FF;
    vpn1 = (va >> 21) & 0x1FF;
    vpn0 = (va >> 12) & 0x1FF;

    // 处理第一级页表
    uint64_t *pte2 = &pgtbl[vpn2];
    uint64_t *pgtbl_lvl1;
    if (!(*pte2 & PTE_V)) {
      pgtbl_lvl1 = (uint64_t *)(kalloc() - PA2VA_OFFSET);
      *pte2 = ((uint64_t)pgtbl_lvl1 >> 12 << 10) | PTE_V;
    }
    pgtbl_lvl1 = (uint64_t *)(((*pte2 >> 10) << 12) + PA2VA_OFFSET);

    // 处理第二级页表
    uint64_t *pte1 = &pgtbl_lvl1[vpn1];
    uint64_t *pgtbl_lvl0;
```

```
      if (!(*pte1 & PTE_V)) {
        pgtbl_lvl0 = (uint64_t *)(kalloc() - PA2VA_OFFSET);
        *pte1 = ((uint64_t)pgtbl_lvl0 >> 12 << 10) | PTE_V;
      }
      pgtbl_lvl0 = (uint64_t *)(((*pte1 >> 10) << 12) + PA2VA_OFFSET);

      // 处理第三级页表
      uint64_t *pte0 = &pgtbl_lvl0[vpn0];
      *pte0 = (((uint64_t)pa & 0x003fffffffffc00) >> 2) | perm;

      // 下一个页面
      va += PGSIZE;
      pa += PGSIZE;
    }
  }
```

- `setup_vm_final` 函数
  - 首先我们需要定义 `_stext` ，`_srodata` ，`_sdata` ，并设置为 `extern`
  - 之后从 `_stext` 地址开始，依次设置va，pa，sz并设置正确的权限位即可

```
/* 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 char _stext[];
extern char _srodata[];
extern char _sdata[];

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

  // No OpenSBI mapping required
  // mapping kernel text X|-|R|V
  uint64_t va = _stext;
  uint64_t pa = (uint64_t)_stext - PA2VA_OFFSET;
  create_mapping((uint64_t *)swapper_pg_dir, va, pa,
                 (uint64_t)(_srodata - _stext), 11);

  // mapping kernel rodata -|-|R|V
  va += _srodata - _stext;
  pa += _srodata - _stext;
  create_mapping((uint64_t *)swapper_pg_dir, va, pa,
                 (uint64_t)(_sdata - _srodata), 3);

  // mapping other memory -|W|R|V
  va += _sdata - _srodata;
```

```
    pa += _sdata - _srodata;
    create_mapping((uint64_t *)swapper_pg_dir, va, pa,
                   PHY_SIZE - (uint64_t)(_sdata - _stext), 7);

    // set satp with swapper_pg_dir
    uint64_t now_satp =
        (((uint64_t)swapper_pg_dir - PA2VA_OFFSET) >> 12) | ((uint64_t)0x8 <<
  60);
    csr_write(satp, now_satp);
    // flush TLB
    asm volatile("sfence.vma zero, zero");
    printk("..setup_vm_final done\n");
    return;
  }
```

4. 编译测试：`make run` 后得到以下结果，说明正确

```
..setup_vm done!
...mm_init done!
..setup_vm_final done
...task_init done!
2024 ZJU Operating System
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]
```

## 二、思考题

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

- 我们将 `main.c` 修改成如下，在 `start_kernel` 中加入输出 `_stext` 和 `_srodata`

```c
extern char _stext[];
extern char _srodata[];

int start_kernel() {
  printk("2024");
  printk(" ZJU Operating System\n");
  printk("The value of _stext is: %lx\n", (uint64_t)(_stext));
  printk("The value of _srodata is: %lx\n", (uint64_t)(_srodata));

  test();
  return 0;
}
```

运行得到以下截图，说明成功设置地址

```
Boot HART MEDELEG         : 0x0000000000f0b509
..setup_vm done!
...mm_init done!
..setup_vm_final done
...task_init done!
2024 ZJU Operating System
The value of _stext is: ffffffe000200000
The value of _srodata is: ffffffe000203000
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
```

- 之后再修改成以下，验证读写属性

```
extern char _stext[];
extern char _srodata[];

int start_kernel() {
  printk("2024");
  printk(" ZJU Operating System\n");
  printk("The value of _stext is: %lx\n", (uint64_t)(*_stext));
  printk("The value of _srodata is: %lx\n", (uint64_t)(*_srodata));
  *_stext = 0x0;
  *_srodata = 0x1;

  test();
  return 0;
}
```

运行得到以下内容，我们根据前两行输出得到 `_stext` 和 `_srodata` 可读，同时观察下两行 `scause` 为f，即15，查询特权级指令手册，我们知道首位为0代表是异常，同时15对应的报错是 `Store/AMO page fault`，由于可读取我们知道这时的page fault是由于不可写导致的。

```
2024 ZJU Operating System
The value of _stext is: 17
The value of _srodata is: 2e
scause: f, sepc: ffffffe0002012d4
scause: f, sepc: ffffffe0002012d4
```

- 接下来我们需要验证这两个字段的可执行性。
  - `_stext` 段我们设置的是可执行，此段为代码段，通常用来存放程序执行代码，由于我们的程序可以正常运行，因此说明此段可执行
  - `_srodata` 段对应存储的是只读数据，因此我们设置的是不可执行，为了验证这一段确实不可以执行，我们同样对 `main.c` 进行如下修改

```
int start_kernel() {
  printk("2024");
  printk(" ZJU Operating System\n");
  //   printk("The value of _stext is: %lx\n", (uint64_t)(*_stext));
  //   printk("The value of _srodata is: %lx\n", (uint64_t)(*_srodata));
  //   *_stext = 0x0;
  //   *_srodata = 0x1;
  asm volatile("call _srodata");

  test();
  return 0;
}
```

运行后得到以下信息

```
..setup_vm_final done
...task_init done!
2024 ZJU Operating System
scause: c, sepc: ffffffe000203000
```

`scause` 为c，即12，同样查阅特权级指令手册知道，代表 `Instruction page fault`，说明PC错误，即执行地址错误，说明我们访问到了不可执行的部分，说明设置正确。

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：

   1. 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page fault，如果不做等值映射，当我们在 `relocate` 中设置 `satp` 后，这之后的输入都会被当成虚拟地址转换成物理地址后执行。这时在 `csrw` 下一行的指令，由于其pc地址为上一条指令，即 `csrw` 指令的pc加4，仍然是物理地址，如果不做等值映射，访问这一页就会出现page fault，导致程序无法继续执行。
   - 原因：在启用分页机制之前，CPU 无法直接使用虚拟地址访问物理地址。因此，通过建立等值映射，可以使得虚拟地址空间的某一部分直接映射到相同的物理地址空间，使得可以找得到这部分物理地址。

   2. 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；

   - 首先我们打开 `arch/riscv/kernel/head.S`，在 `_start` 中，首先进行了一些初始化操作，如屏蔽所有中断、加载全局指针(GF)、禁用浮点单元、主处理器选择、清空 `.bss` 段、保存硬件线程等操作。
   - 之后，采用和我们在之前实验中类似的操作调用 `setup_vm` 和 `relocate` 设置虚拟内存，之后恢复栈指针和 `tp`，最后启动内核。

```
/* 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
    call parse_dtb
    tail start_kernel
```

- 之后我们首先来分析 `setup_vm`，见以下注释内容，主要完成了偏移量的计算以及使用三级页表还是四级页表

```
asmlinkage void __init setup_vm(void)
{
    // 函数及变量定义
    extern char _start;
    uintptr_t i;
    uintptr_t pa = (uintptr_t) &_start;
    pgprot_t prot = __pgprot(pgprot_val(PAGE_KERNEL) | _PAGE_EXEC);  // 权限
    // 计算虚拟地址和物理地址的偏移
    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);
     // 设置页表结构，判断是否使用PMD折叠，使用三级页表还是四级页表
#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
}
```

- 接下来我们分析 `relocate` 函数

```
relocate:
    /* Relocate return address */
    #计算虚拟地址与物理地址的偏移，以调整ra的值
    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 */
    #这里将stvec的地址从物理地址转换为虚拟地址，这一步很关键。这是因为Linux没有进行等值映射。那么在我们通过设置satp启用MMU后，以下所有指令都会被当成虚拟地址传入MMU转换成物理地址进行执行。但此时trap_handler的地址，以及satp的下一条指令还仍然是物理地址，就会出现page fault。这时，为了触发page fault可以正确地进入trap_handler，自然我们需要设置stvec
    la a0, 1f
    add a0, a0, a1
    csrw CSR_STVEC, a0

    /* Compute satp for kernel page tables, but don't load it yet */
    #计算swapper_pg_dir页表的虚拟地址，和实验中写的类似
    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.
     */
     #设置线性页表并设置satp
    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

    ret
```

3. 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首先在 `reloacte` 中将 `stvec` 加上偏移量。
- 之后在执行设置完 `satp` 后的下一条指令，由于此时的地址仍然是物理地址，在传入MMU后无法找到对应的地址，就会产生page fault。这时候由于我们将 `stvec` 加上偏移量了，这时候我们就可以正常访问 `trap_handler` 。此时我们只需要在 `trap_handler` 内将 `sepc` 加上虚拟地址即可。这样我们跳出 `trap_handler` 程序后，就可以返回到正确的地址。

4. 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` 是一个用于内核启动阶段的临时页表，通常用于处理内核启动时的虚拟地址映射。在这个阶段，内核可能还没有完全切换到常规的内核虚拟地址空间，因此需要一个简单的页表来支持早期的代码执行，允许内核在虚拟地址空间和物理地址空间之间做必要的切换。

  设置：

```
#设置线性页表并设置satp
    la a0, trampoline_pg_dir
    srl a0, a0, PAGE_SHIFT
    or a0, a0, a1
    sfence.vma
    csrw CSR_SATP, a0
```

- `swapper_pg_dir` 是内核的主要页表，包含了内核虚拟地址空间的页表映射。`swapper_pg_dir` 被用于系统的正常运行阶段，在内核启动完成后，内核将切换到使用这个页表来管理内存映射。也就是我们所使用的三级页表

  设置：

```
/* Compute satp for kernel page tables, but don't load it yet */
    #计算swapper_pg_dir页表的虚拟地址，和实验中写的类似
    la a2, swapper_pg_dir
    srl a2, a2, PAGE_SHIFT
    li a1, SATP_MODE
    or a2, a2, a1
    ...
.align 2
1:
    ...
.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

    ret
```

5. 尝试修改你的 kernel，使得其可以像 Linux 一样不需要等值映射。
   - 首先在 `relocate` 中设置 `stvec` 为我们新设置的 `_new_traps`

```
la a0, _new_traps
add a0, a0, t0
csrw stvec, a0
```

- 之后书写 `_new_traps`，在函数中，我们只需要将 `sepc` 加上虚拟地址和物理地址之间的偏移量即可，这样我们就可以返回到正确的地址了

```
    .globl _new_traps
_new_traps:
    addi sp,sp,-8
    sd t0, 0(sp)

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

    ld t0,0(sp)
    addi sp,sp,8
    sret
```

- 最后值得注意的是，由于我们只在第一次才需要跳转进 `_new_traps`，我们需要在 `head.S` 中调用 `relocate` 后，立即设置 `stvec` 为 `trap_handler`，避免出错

```
jal setup_vm
jal relocate

# set stvec = _traps
la a0, _traps
csrw stvec, a0

jal mm_init
jal setup_vm_final
```

- 这样我们就可以正常运行了

```
Boot HART MEDELEG         : 0x0000000000f0b509
..setup_vm done!
sepc:ffffffe0002000a4...mm_init done!
..setup_vm_final done
...task_init done!
2024 ZJU Operating System
[S] Supervisor Mode Timer Interrupt
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
```