操作系统实验报告

课程名称	:	操作系统
姓　　名	:	沈一芃
学　　院	:	计算机学院
专　　业	:	计算机科学与技术
学　　号	:	3220101827
指导教师	:	李环 柳晴

2024 年　11　月　18　日

Lab 3 Report

实验名称：RV64 虚拟内存管理

联系方式：1669335639@qq.com 18310211513

1. 实验目的与要求

1.1. 实验目的

- 学习虚拟内存的相关知识，实现物理地址到虚拟地址的切换
- 了解 RISC-V 架构中 SV39 分页模式，实现虚拟地址到物理地址的映射，并对不同的段进行相应的权限设置

1.2. 实验要求

- 完成等值映射和direct mapping的页表映射
- 正确设置satp寄存器并启动虚拟地址，使操作系统运行在虚拟地址上。
- 完成SV39三级页表映射
- 正确设置satp寄存器，使操作系统运行在SV39页表的虚拟地址映射上。

2. 实验环境

本机CPU	本机操作系统	实验环境	实验操作系统	配置环境
Mac Silicon M3	CentOS Sequoia 15.0.1	Docker 4.33.0	Ubuntu 24.10	Qemu-system-riscv64 9.0.2

3. 实验过程

3.1. 实验环境搭建

- 整个实验建立在lab2的基础上。从仓库克隆实验2代码到本地，跟随实验流程搭建文件结构如下：

```
├── arch
│   └── riscv
│       ├── include
│       │   ├── mm.h        # need to modify; change the address
│       │   ├── proc.h
```

```
│       │   ├── clock.h
│       │   ├── defs.h      # need to modify; add macro given by the storage
│       │   └── sbi.h
│       ├── kernel
│       │   ├── vm.c        # need to accomplish; set up the phy-vir address mapping
│       │   ├── mm.c
│       │   ├── proc.c
│       │   ├── sbi.c
│       │   ├── trap.c
│       │   ├── clock.c
│       │   ├── head.S      # need to accomplish; set the CSR_satp to activate virtual
address
│       │   ├── entry.S
│       │   ├── Makefile
│       │   └── vmlinux.lds # need to modify; directly pull the new vmlinux.lds from the
lab3 storage
│       └── Makefile
├── include
│   ├── printk.h
│   ├── stddef.h
│   └── stdint.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── printk.c
└── Makefile                # need to modify; add a compiling instruction to forbid PIE
```

- `defs.h`中添加宏定义如下：

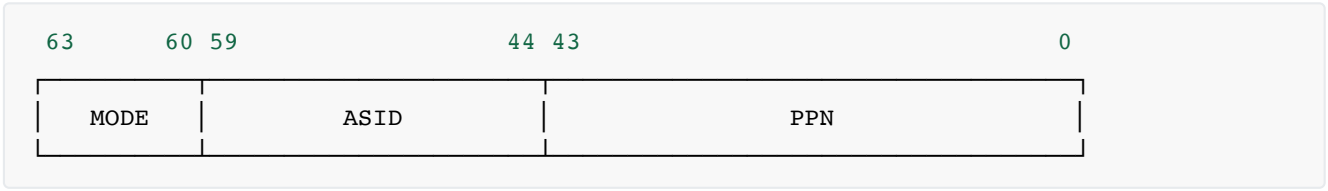
```
#define OPENSBI_SIZE (0x200000)

#define VM_START (0xffffffe000000000)
#define VM_END (0xffffffff00000000)
#define VM_SIZE (VM_END - VM_START)

#define PA2VA_OFFSET (VM_START - PHY_START)
```

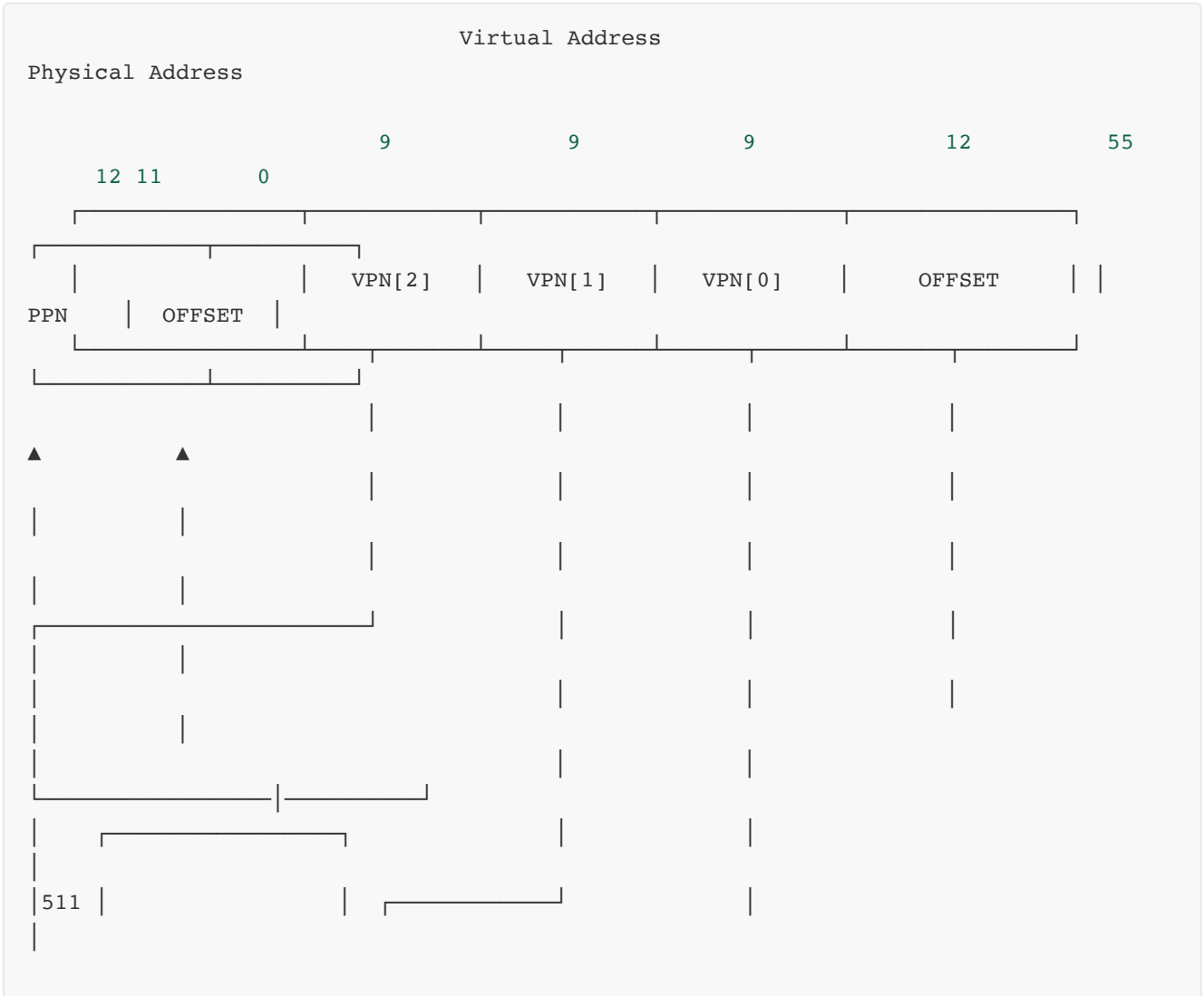
- `Makefile`的`CF`中添加`-fno-pie`以关闭PIE：
  - PIE，位置无关执行，通过 GOT 表来取地址，使得代码被放在不同地址执行，也可以取到正确的地址。
  - 但在我们的kernel中，所有地址保持不变，始终被 load 到 0x80200000 ，因此这个 PIE 对我们并没有作用。
  - 此外，本次实验中，由于开启虚拟地址，PIE的GOT还会有副作用；因为 GOT 表里的地址都是最终的虚拟地址，所以在 kernel 启用虚拟地址之前，从 GOT 表取出的地址都是错的。开启PIE会导致操作系统开启虚拟地址前无法获取正确的地址，从而无法启动运行。

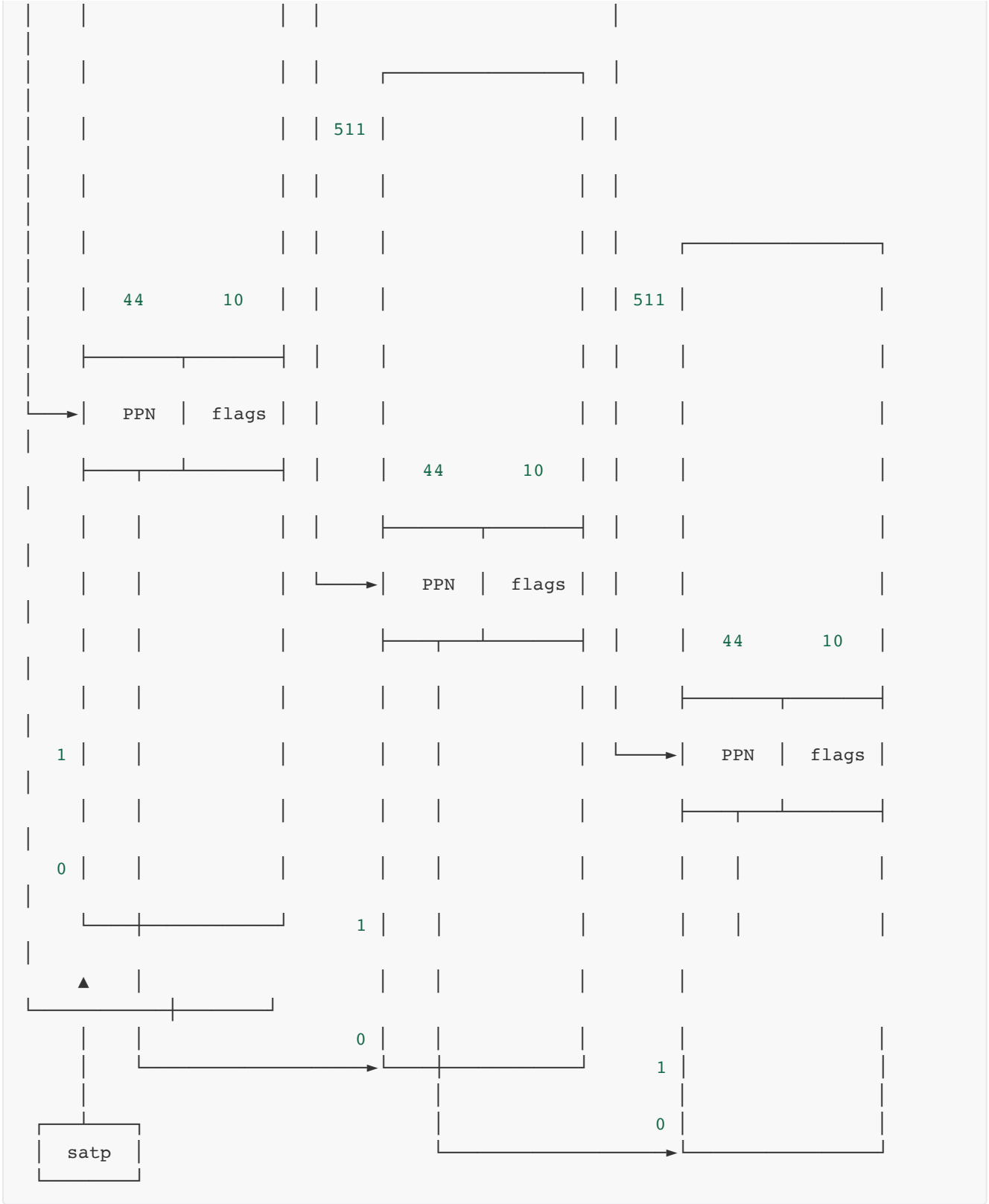
3.2. RV64 内核

3.2.0. 总括

- 本次实验的目的是开启虚拟地址，让操作系统运行在虚拟地址上。这样的好处是，为每个进程提供连续的内存空间，隐藏了物理内存的复杂性；保证不同进程的内存隔离，方便内存的取入取出和管理。在下一个实验中要引入用户态进程，会使内存分配和管理愈发复杂，故而在本次实验我们要实现虚拟地址。
- 虚拟地址本质是一块存在于逻辑上的，大小远超实际物理内存的空间。这样就可以给操作系统代码和进程的代码各自分配足够的连续、独立的空间。但众所周知，不论是操作系统还是进程，实际运行时要在物理内存中进行。虽然我们在编程/管理时只需要对着虚拟地址操作，但虚拟内存中的每一个地址要对应到实际物理内存中的地址；这就需要一个映射关系来保证虚拟地址和物理地址的对应（想来这个从虚拟地址到物理地址的映射不一定是单射也不一定是满射）。这个映射关系靠页表记录。
- 对本次实验而言，satp寄存器决定了操作系统是运行在物理地址还是虚拟地址。该寄存器结构如下，通过设置MODE可以选择是否采用虚拟地址/采用的映射方式。本次实验最终要实现采用MODE=d8时对应的SV39三级页表虚拟地址映射方式。该方式流程见下图，较为直接，不再赘述。







- 所以，实验的流程可以理解为，先正确建立页表中的映射，然后设置satp寄存器开启虚拟地址。然而，本次实验并非直接建立三级页表并设置satp；此处的流程是先建立一个简单的direct mapping虚拟页表（至于具体实现，有两种方式；一种是如实验流程，还要借助等值映射；另一种方式则如思考题2，不借助等值映射。原因也参见思考题），启动虚拟内存。当操作系统运行在虚拟内存后，再建立复杂的SV39三级页表，重新设置satp寄存器，使程序运行在SV39的虚拟页表上。这个的原因参加心得体会部分。

3.2.1. `setup_vm`实现简单页表

- 这一部分就是建立一个简单的页表，先让整个虚拟内存运行在虚拟内存上；而这个简单的页表就是direct mapping，直接将每一个物理地址映射到偏置的direct mapping area，即`PA + PA2VA_OFFSET == VA`。

> 此处建议阅读RISC-V-privilege手册的10.3.2节，其中有RV32页表的完整寻址流程。

- 虽然我们建立的是一个direct mapping，但MODE还是设置为8，即SV39三级页表模式。但我们实际上只需要satp和VPN2这一级页表，将剩下的部分全部作为offset即可。这依赖了上述该节中SV类页表的映射方式：当页表项prot位中X/R/W存在一位是1，则认为该页表是叶子页表。在这一部分，我们直接将VXRW四位全部置1.
- 然而，在这一部分，我们不光进行了direct mapping，还要进行另一个等值映射，即`PA == VA`，流程同direct mapping，区别仅在于做映射的虚拟地址不同。

> 这样做的原因参见思考题2

- 实现代码如下，请注意区分虚拟地址、物理地址、页表项的格式！注意页表的序号是从**第10位**开始还是从**第12位**开始。

```
    memset(early_pgtbl, 0, sizeof(early_pgtbl));

    uint64_t addr, vaddr;
    addr = PHY_START;
    vaddr = addr;
    index = (vaddr >> 30) & 0x1FF;
    early_pgtbl[index] = (((addr >> 30) & 0x1ff) << 28) | PTE_V | PTE_R | PTE_W | PTE_X;
    vaddr = addr + PA2VA_OFFSET;
    index = (vaddr >> 30) & 0x1FF;
    early_pgtbl[index] = (((addr >> 30) & 0x1ff) << 28) | PTE_V | PTE_R | PTE_W | PTE_X;
```

3.2.2. `relocate`设置satp启动虚拟内存

- 在`_start`中，设置完sp后调用`setup_vm`和`relocate`，分别建立页表并设置satp启动虚拟地址。
- `relocate`的逻辑就是用CSRW指令写satp寄存器，从而启动虚拟地址。satp的结构上文已经提及，ppn部分即`setup_vm`中建立的页表的地址。在写satp前后，要调用`sfence.vma zero, zero`指令，这条指令用于维护内存的一致性，做法是确保修改的页表项生效，并清空TLB。关于写satp和sfence的关系，具体详见实验指导手册。但无论如何，在写后加上一条肯定不会错且有保证。

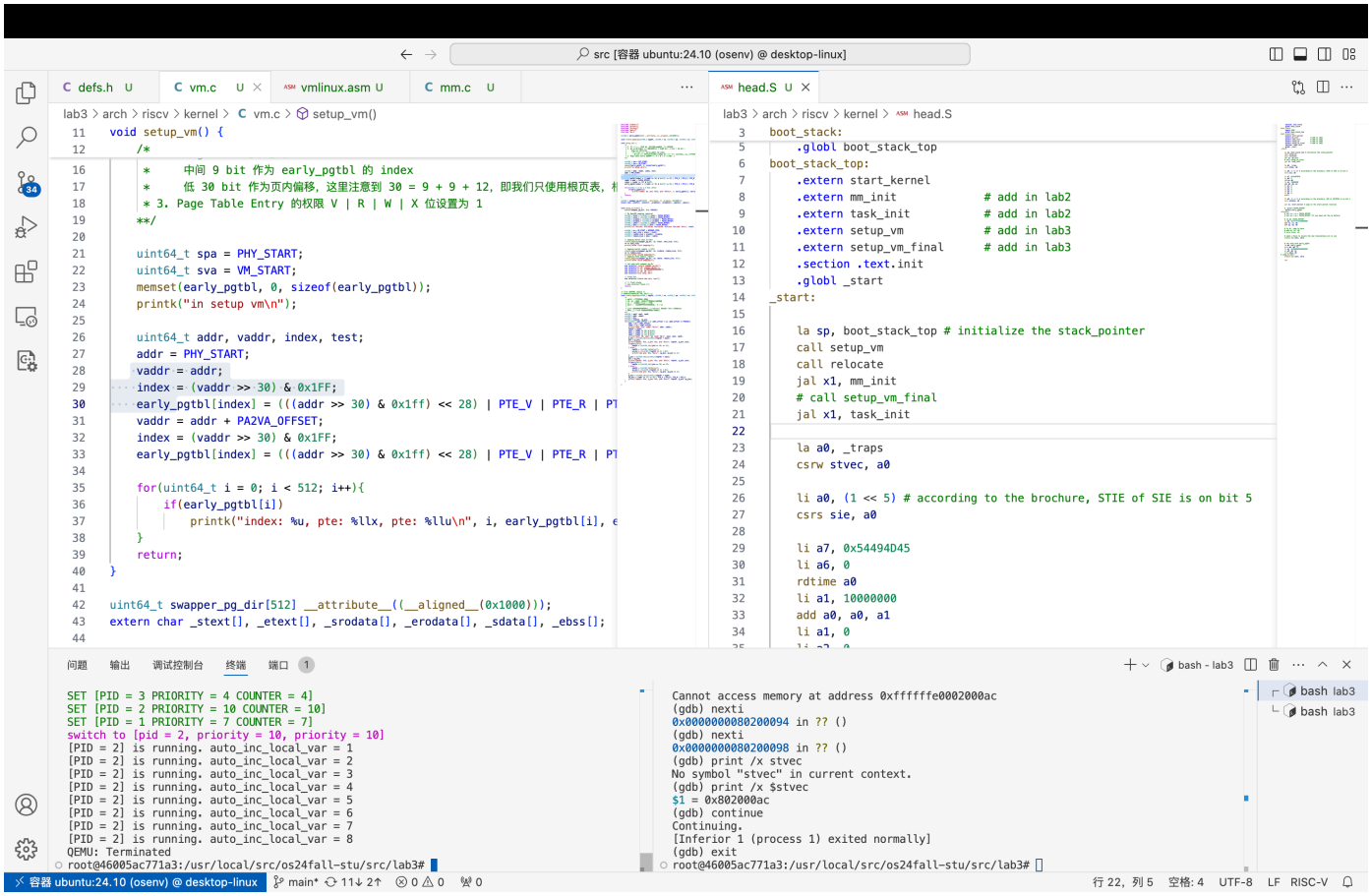
> 有关TLB：作为一个软工班里仅有的cs学生，终于有一个我学过但是他们没学过的概念了。。。虽然没什么用但讲一下，这就类似于一个快查表，优先级高于页表。用三级页表查还要跳三次，但这个表直接从虚拟地址到物理地址一步到位，其中记录的也想来是那些很常用的地址。如果更新了页表但没有刷新TLB，遇到TLB中有存储的虚拟地址，会优先按照TLB中找物理地址，有可能发生页表中修改了映射但此处仍是旧映射的问题。

- 然而，观察预设代码，会发现在`relocate`部分还需要更新sp和ra。sp的更新较好理解，至于ra为什么要更新，其实和等值映射以及思考题有密切联系。

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

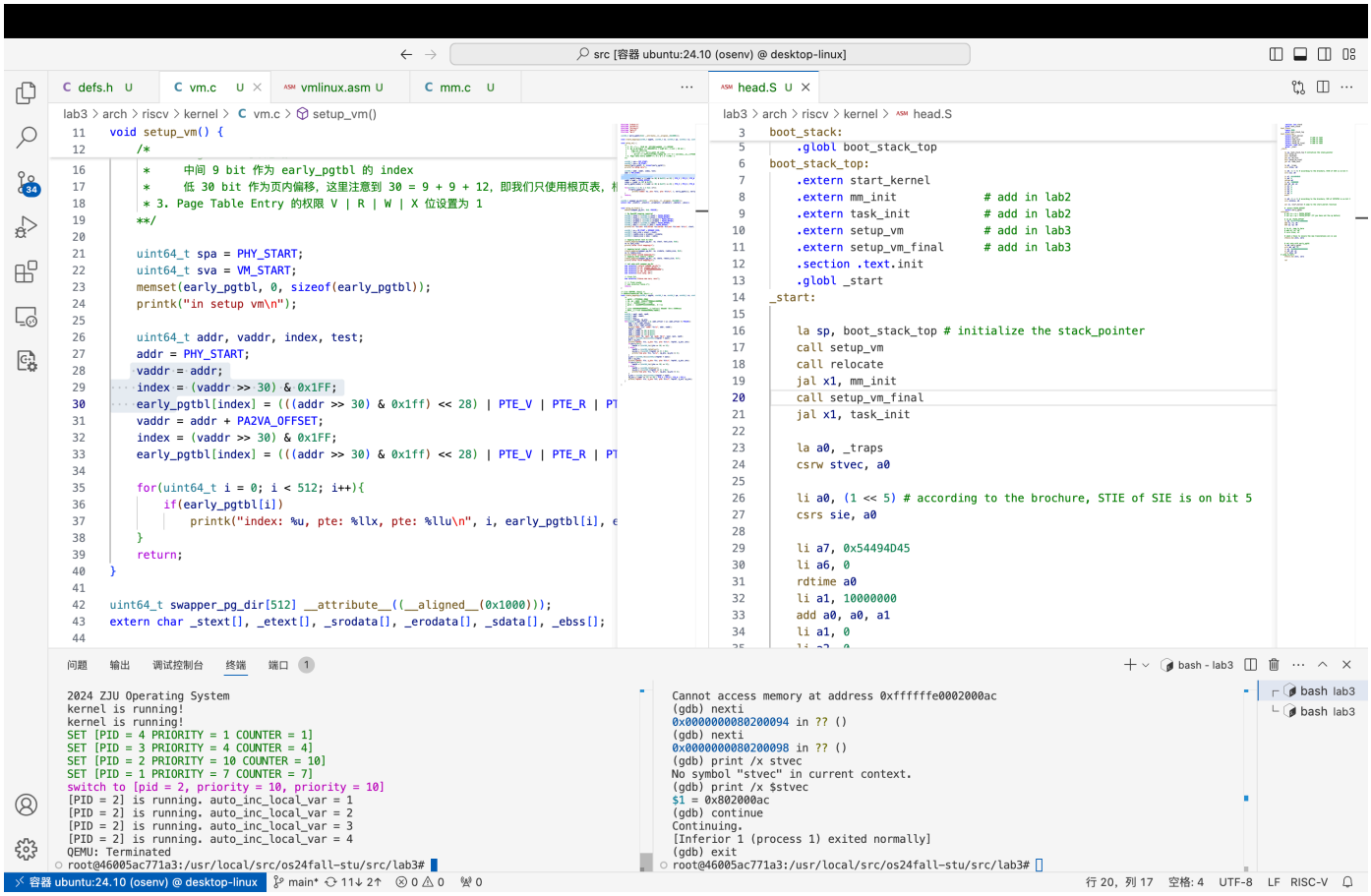
```
    li a0, 0xffffffdf80000000
    add ra, ra, a0
    add sp, sp, a0

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

    # set satp with early_pgtbl
    la a0, early_pgtbl
    srli a0, a0, 12
    li a1, 0x8000000000000000
    or a0, a0, a1
    csrw satp, a0
    sfence.vma zero, zero

    ret
```

- 完成后终端`make run`，操作系统已经可以正确运行在虚拟地址上。



3.2.3. `setup_vm_final`建立三级页表并更新satp

- 操作系统已经运行在了虚拟内存上。目前我们要完整建立三级页表，让操作系统运行在SV39三级页表的映射上。
- 这一段调用`creat_mapping`，对va开始的sz范围内的页进行映射。
- 在最后记得更新satp并刷新。



4. 讨论与心得

4.1. 分两次建立页表

- 本次实验有一个很困扰我的地方，在于为什么要分两次建立页表。以下我做出两种解释。
  - 首先，完全可以直接建立三级页表并设置satp,一次性完成，尤其对于本次实验这种较为简单的结构。
  - 我个人认为，其中的关键在于对虚拟内存的分配。
    - direct mapping和三级页表的区别在于，direct mapping只需要一级页表，而这一级页表的空间是操作系统内核自己声明页表，在系统内核的bss段栈内分配好的。详见`system map`中，direct mapping的页表地址是`0x80208000`到`0x80209000`，而bss段的空间是`0x8050000`到`0x8090000`，正好到`_ekernel`，也符合一开始分配的4kib大小的空间。
    - 但如果要直接使用三级页表，对于二级或者三级页表，需要主动申请一页空间。这部分的内容见`mm.c`。然而，使用`kalloc`进行内存分配依赖于`kfreerange`对指定范围内页空间的回收，指定范围则是从`_ekernel`开始。但`vmlinux.lds`的链接布局是按照虚拟地址进行设计的。故而对于非操作系统内核部分的内存管理分配要在操作系统已经运行在虚拟内存后才能正确工作。这就导致无法为三级页表分配新的kalloc页。当然，这只是当前这种设计的局限性。如果要实现一次性建立三级页表并设置satp，对于这个简单的内核，只需要想办法处理好mm.c中的地址问题，将其先设置为物理地址，放在建立页表前初始化内存，再在开启虚拟内存后切换至虚拟地址即可。但这样就很麻烦了，况且真正的linux内核功能更全面，结构更复杂；再加上用户态进程，这样切换显然不如先用简单的方式让内核运行在虚拟地址上，再初始化内存，再建立三级页表方便。

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

  - 接下来的这个观点来自于我的室友`Matilda`。据他查阅资料，linux源码的某次提交中有讨论过这个问题。当物理内存空间过大，导致物理内存末尾和虚拟内存开始处出现冲突时，会导致系统无法区分物理内存和虚拟内存的情况。具体原因还没想清楚，或许以后会继续深究。

4.2. 调试方法

- 本次实验中，内核在第一次设置satp前运行在物理地址上，之后运行在虚拟地址上。故而，在调试`relocate`的`csrw satp`之前要用`0x8020....`进行调试，在这之后应该用`0xffffffe00020....`进行调试。

5. 思考题

5.1. 验证 `.text`，`.rodata` 段的属性

-

打印页表项如下

```
addr: 80200000, vaddr: ffffffe000200000
vpn2: 384, vpn1: 1, vpn0: 0
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 0
new pte: 3ffffff801fffc01, ffffffe007fff004
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 0
new pte: 3ffffff801fff801, ffffffe007ffe004
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe000, pte: 2008000b
addr: 80201000, vaddr: ffffffe000201000
vpn2: 384, vpn1: 1, vpn0: 1
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe001, pte: 2008040b
addr: 80202000, vaddr: ffffffe000202000
vpn2: 384, vpn1: 1, vpn0: 2
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe002, pte: 2008080b
after first mapping
addr: 80203000, vaddr: ffffffe000202484
vpn2: 384, vpn1: 1, vpn0: 2
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe002, pte: 20080c03
after second mapping
addr: 80204000, vaddr: ffffffe00020277c
vpn2: 384, vpn1: 1, vpn0: 2
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe002, pte: 20081007
addr: 80205000, vaddr: ffffffe00020377c
vpn2: 384, vpn1: 1, vpn0: 3
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe003, pte: 20081407
addr: 80206000, vaddr: ffffffe00020477c
vpn2: 384, vpn1: 1, vpn0: 4
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe004, pte: 20081807
addr: 80207000, vaddr: ffffffe00020577c
vpn2: 384, vpn1: 1, vpn0: 5
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe005, pte: 20081c07
addr: 80208000, vaddr: ffffffe00020677c
vpn2: 384, vpn1: 1, vpn0: 6
tpmtbl: ffffffe000208000, p_pte: ffffffe000208180, pte: 3ffffff801fffc01
tpmtbl: ffffffe007fff000, p_pte: ffffffe007fff001, pte: 3ffffff801fff801
tmptbl: ffffffe007ffe000, p_pte: ffffffe007ffe006, pte: 20082007
after third mapping
...task_init done!
2024 ZJU Operating System
```

5.2. 探索等值映射的原因

5.2.1. 为什么要等值映射

- 对于这个问题，将关注点放在`relocate`写satp的`csrw`指令后。当开启satp后，操作系统运行在虚拟内存上，所有的地址都按页表进行查询。然而，**satp指令并不会改变pc的值**。在`3.2.2`节中提及，要注意为什么设置了sp的同时设置了ra。其实本次实验的内核就是**依靠ra来将pc从物理地址修改为虚拟地址**的。但这样就暴露了一个问题，在运行csrw后到ret和ret之前的指令时，pc仍然在物理地址运行；而虚拟地址的转化已经开启。
- 这就导致，我们需要在pc的值是物理地址时，仍能通过页表将其映射到正确的物理地址。这就是等值映射的意义。实际上。如果能精准定位这一部分代码，只对这几行代码做等值映射也就足够达成目标。

5.2.2. linux内核虚拟地址启动逻辑

- linux 5.2.21源码中`/arch/riscv/kernel/head.S`和`/arch/riscv/mm/init.c`的代码是关键。其实整个流程和我们的内核极其相似，先阻塞interrupt中断，再清理段内空间，之后用setup_vm初始化了trampoline和swapper两个页表，在relocate中更新各种寄存器，如ra，sp，**stvec**，satp。这里也是先用trampoline页表启动虚拟内存，再用swapper页表更新系统运行的三级页表。

5.2.3. linux内核解决等值映射

- 如上所言，关键点在于**stvec**的更新。stvec用于触发trap时，决定跳回何处。5.2.1中分析了不做等值映射的问题，将一个物理地址当作虚拟地址进行映射，无法访问对应的页。这会使系统抛出page fault。我们只需要利用这个错误的抛出，通过stvec将pc更改至正确的地址，就可以实现不通过等值映射解决pc过渡的问题。

5.2.4. trampol dir和swapper dir

- `trampoline_pg_dir`是临时页表，只映射内核所在的物理内存的前几个超级页，作为过渡，范围很小。
- `swapper_pg_dir`是正式页表，映射整个内核地址空间，完整，功能齐全，范围大。
- 内核代码中，将`trampol_pg_dir`存入a2，将`swapper_pg_dir`存入a0，随后先将a0写入satp，之后又一次更新stvec，再将a2写入satp。关键代码部分如下：

```
/* 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

  ret
```

5.2.5. 实现不用等值映射的系统内核启动

- 仿照linux v5.2.21写stvec策略即可。