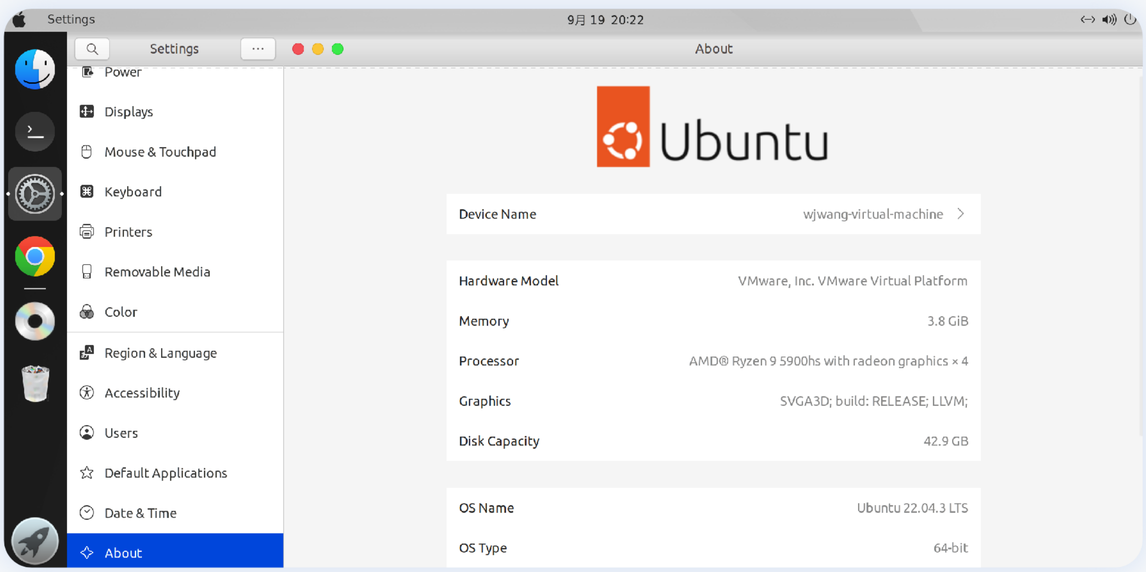
Lab 3 RV64 虚拟内存管理

学号：3210106034
姓名：王伟杰

> 实验环境：
>
> ```
> uname -a
> Linux wjwang-virtual-machine 6.2.0-33-generic #33~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Thu Sep  7 10:33:52 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
> ```
>
> 
>

准备工程

在 `defs.h` 添加如下内容

```c
#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)
```

从 `repo` 同步 `vmlinux.lds`，并正确放置。

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：

```makefile
# Makefile
# ...
ISA=rv64imafd_zifencei
# ...
all: clean
	${MAKE} -C lib all
	${MAKE} -C test all
	${MAKE} -C init all
	${MAKE} -C arch/riscv all
	@echo -e '\n'Build Finished OK
# ...
```

开启虚拟内存映射

`setup_vm` 的实现

对 `PHY_START` 开始的1GB区域进行两次映射：一次等值映射与一次映射到 `direct mapping area`。

由于 `PA` 的格式如下：

```
 55                30 29        21 20        12 11                           0
 -----------------------------------------------------------------------------
|       PPN[2]       |   PPN[1]   |   PPN[0]   |          page offset          |
 -----------------------------------------------------------------------------
                            Sv39 physical address
```

我们不需要使用多级页表，所以 `PPN` 的值即为上图中的 `PPN[2]` ， `PTE` 格式如下：

```
 63      54 53        28 27        19 18        10 9   8 7 6 5 4 3 2 1 0
 -----------------------------------------------------------------------
| Reserved |    PPN[2]    |    PPN[1]    |    PPN[0]    | RSW |D|A|G|U|X|W|R|V|
 -----------------------------------------------------------------------
                                                     ↑   ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
                                                     |   | | | | | | | `---- V - Valid
                                                     |   | | | | | | `------ R - Readable
                                                     |   | | | | | `-------- W - Writable
                                                     |   | | | | `---------- X - Executable
                                                     |   | | | `------------ U - User
                                                     |   | | `-------------- G - Global
                                                     |   | `---------------- A - Accessed
                                                     |   `------------------ D - Dirty
                                                     `---------------------- Reserved for supervisor software
```

所以我们将得到的 `PPN` 左移28位，再设置权限位即可。

```c
// arch/riscv/kernel/vm.c
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 PA = PHY_START;
    unsigned long VA_EQ = PA;
    int index = (VA_EQ >> 30) & 0x1ff;
```

```c
    unsigned long PPN = PA >> 30 & 0x3ffffff;
    unsigned long PTE = (PPN << 28) | 0xf; // V R W X = 1
    early_pgtbl[index] = PTE;

    unsigned long VA_DIRECT = PA + PA2VA_OFFSET;
    index = (VA_DIRECT >> 30) & 0x1ff;
    PPN = PA >> 30 & 0x3ffffff;
    PTE = (PPN << 28) | 0xf;
    early_pgtbl[index] = PTE;

    printk("...setup_vm done!\n");
}
```

这里还需要更改设置 `sp` 到 `boot_stack` 的顶部的代码，因为 `vmlinux.lds` 将kernel代码起始位置修改为了 `VM_START + OPENSBI_SIZE`：

```asm
# arch/riscv/kernel/head.S
# ...
_start:
    # ------------------
    # - your code here -
    # 设置sp到boot_stack的顶部
    li t0, 0xffffffdf80000000
    la sp, boot_stack_top
    sub sp, sp, t0

    call setup_vm
    call relocate

    call mm_init
    call task_init
# ...
```

`satp` 寄存器（`PA >> 12 == PPN`）：

```
 63      60 59                  44 43                                0
 ---------------------------------------------------------------------
|   MODE   |        ASID         |                PPN                |
 ---------------------------------------------------------------------
```

同时我们完成 `relocate` 函数，完成对 `satp` 的设置，以及跳转到对应的虚拟地址：

```
relocate:
    # set ra = ra + PA2VA_OFFSET
    # set sp = sp + PA2VA_OFFSET (If you have set the sp before)
    # PA2VA_OFFSET = 0xffffffe000000000 - 0x0000000080000000 =
0xffffffdf80000000
    li t1, 0xffffffdf80000000
    add ra, ra, t1
    add sp, sp, t1

    # set satp with early_pgtbl
    la t0, early_pgtbl
    sub t0, t0, t1
    srl t0, t0, 12
    li t1, 0x8000000000000000
    or t0, t0, t1
    csrw satp, t0

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret
```

`setup_vm_final` 的实现

修改 `mm.c` 中的代码， `mm.c` 中初始化的函数接收的起始结束地址调整为虚拟地址：

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

由于之前映射的页表比较粗糙，并且后续不再需要等值映射，所以我们在这里对所有物理内存 (128M) 进行映射，并设置正确的权限：

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

```c
/* swapper_pg_dir: kernel pagetable 根目录， 在 setup_vm_final 进行映射。 */
unsigned long  swapper_pg_dir[512] __attribute__((__aligned__(0x1000)));
extern char _stext[], _etext[];
extern char _srodata[], _erodata[];
extern char _sdata[], _edata[];
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, uint64 perm);

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

    // No OpenSBI mapping required

    // mapping kernel text X|-|R|V
    create_mapping((uint64 *)swapper_pg_dir, (uint64)_stext, (uint64)_stext - PA2VA_OFFSET, (uint64)(_etext - _stext), 0xb);

    // mapping kernel rodata -|-|R|V
    create_mapping((uint64 *)swapper_pg_dir, (uint64)_srodata, (uint64)_srodata - PA2VA_OFFSET, (uint64)(_erodata - _srodata), 0x3);

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

    // set satp with swapper_pg_dir
    unsigned long satp = 0x8000000000000000 | ((unsigned long)swapper_pg_dir - PA2VA_OFFSET) >> 12;
    csr_write(satp, satp);

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

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

在 `create_mapping` 中设置三级页表映射关系：

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

```c
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, uint64 perm) {
    /*
    pgtbl 为根页表的基地址
    va, pa 为需要映射的虚拟地址、物理地址
    sz 为映射的大小，单位为字节
    perm 为映射的权限（即页表项的低 8 位）

    创建多级页表的时候可以使用 kalloc() 来获取一页作为页表目录
    可以使用 V bit 来判断页表项是否存在
    */
    int page_num = (sz + PGSIZE - 1) / PGSIZE; // 取整
    for (int i = 0; i < page_num; i++) {
        uint64 VPN[3];
        VPN[2] = (va >> 30) & 0x1ff; // 9 bit
        VPN[1] = (va >> 21) & 0x1ff;
        VPN[0] = (va >> 12) & 0x1ff;
        uint64 *pte = pgtbl;
        for (int j = 2; j > 0; j--) {
            if ((pte[VPN[j]] & 0x1) == 0) { // 如果valid为0，则需要开辟新的pte
                uint64 new_pte = kalloc();
                pte[VPN[j]] = (((new_pte - PA2VA_OFFSET) >> 12) & 0xfffffffffff) << 10 | 0x1;
                pte = (uint64 *)new_pte;
            }
            else {
                pte = (uint64 *)((pte[VPN[j]] >> 10 << 12) + PA2VA_OFFSET);
            }
        }
        pte[VPN[0]] = (((pa >> 12) & 0xfffffffffff) << 10) | perm;
        va += PGSIZE;
        pa += PGSIZE;
    }
}
```

编译及测试

```
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...setup_vm done!
...mm_init done!
...proc_init done!
2022 Hello RISC-V
sstatus = 8000000000006002
sscratch = 0
[INTERRUPT] S mode timer interrupt!

switch to [PID = 8 COUNTER = 1 PRIORITY = 25]
[PID = 8] is running. auto_inc_local_var = 1
[INTERRUPT] S mode timer interrupt!

switch to [PID = 17 COUNTER = 1 PRIORITY = 71]
[PID = 17] is running. auto_inc_local_var = 1
[INTERRUPT] S mode timer interrupt!

switch to [PID = 18 COUNTER = 1 PRIORITY = 64]
[PID = 18] is running. auto_inc_local_var = 1
[INTERRUPT] S mode timer interrupt!

switch to [PID = 7 COUNTER = 2 PRIORITY = 5]
[PID = 7] is running. auto_inc_local_var = 1
[INTERRUPT] S mode timer interrupt!
[PID = 7] is running. auto_inc_local_var = 2
[INTERRUPT] S mode timer interrupt!

switch to [PID = 14 COUNTER = 2 PRIORITY = 6]
[PID = 14] is running. auto_inc_local_var = 1
[INTERRUPT] S mode timer interrupt!
[PID = 14] is running. auto_inc_local_var = 2
[INTERRUPT] S mode timer interrupt!
```

思考题

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

   在成功设置之后，我们知道 `.text`， `.rodata` 段是不允许写入的，所以在设置之后对其进行写入，如果可以成功写入则证明没有设置成功。

   在 `setup_vm_final` 函数的末尾加入如下代码，尝试对 `.text`， `.rodata` 段写入：

```c

    printk("...setup_vm_final done!\n");

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

    *_stext = 0x00;
    printk("*_stext = %x\n", *_stext);
    *_srodata = 0x00;
    printk("*_srodata = %x\n", *_srodata);

    return;
```

先注释掉对 `.rodata` 进行操作的部分，运行代码，由于权限不够所以程序暂停运行。

```
make run
Domain0 HARTs              : 0*
Domain0 Region00           : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01           : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address       : 0x0000000080200000
Domain0 Next Arg1          : 0x0000000087000000
Domain0 Next Mode          : S-mode
Domain0 SysReset           : yes

Boot HART ID               : 0
Boot HART Domain           : root
Boot HART ISA              : rv64imafdcsu
Boot HART Features         : scounteren,mcounteren,time
Boot HART PMP Count        : 16
Boot HART PMP Granularity  : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count       : 0
Boot HART MHPM Count       : 0
Boot HART MIDELEG          : 0x0000000000000222
Boot HART MEDELEG          : 0x000000000000b109
...setup_vm done!
...mm_init done!
...setup_vm_final done!
_stext = 0000009b
```

同理，注释掉对 `.text` 段的操作：

```
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...setup_vm done!
...mm_init done!
...setup_vm_final done!
 srodata = 0000002e
```

证明 `.text` ， `.rodata` 段的属性已经成功设置。

2. 为什么我们在 `setup_vm` 中需要做等值映射?

   因为在 `relocate` 执行过程中，虽然会设置 `satp` 寄存器的值，但是在设置完之后， `pc` 仍然是物理地址，所以需要在原本的位置做一次等值映射，保证这个函数仍然可以运行这个位置的代码，使之顺利运行完。由于我们设置过 `ra` ，所以这个函数运行完之后会返回到虚拟地址继续运行。

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。

   查找Linux源码，Linux是这样写relocate函数的：

```
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 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
     */
    la a0, trampoline_pg_dir
    srl a0, a0, PAGE_SHIFT
    or a0, a0, a1
    sfence.vma
    csrw sptbr, a0
1:
    /* Set trap vector to spin forever to help debug */
    la a0, .Lsecondary_park
    csrw stvec, a0
```

从中我们获得灵感，可以从 `exception` 入手调整 `pc` 在向虚拟地址转换过程中找不到正确路径的问题，首先我们删除 `setup_vm` 中等值映射的代码：

```c
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 PA, VA_EQ, VA_DIRECT, PPN, PTE;
    int index;
    PA = PHY_START;
    // VA_EQ = PA;
    // index = (VA_EQ >> 30) & 0x1ff;
    // PPN = PA >> 30 & 0x3ffffff;
    // PTE = (PPN << 28) | 0xf; // V R W X = 1
    // early_pgtbl[index] = PTE;

```

```c
    VA_DIRECT = PA + PA2VA_OFFSET;
    index = (VA_DIRECT >> 30) & 0x1ff;
    PPN = PA >> 30 & 0x3ffffff;
    PTE = (PPN << 28) | 0xf;
    early_pgtbl[index] = PTE;

    printk("...setup_vm done!\n");
}
```

然后在 `relocate` 函数中加入处理 `pc` 的错误跳转，在转换时trap to `stvec` 到标签为 `1` 的位置：

```asm
relocate:
    # relocate return address
    li t1, 0xffffffdf80000000
    add ra, ra, t1
    add sp, sp, t1

    # point stvec to virtual address of instruction after satp write
    la t0, 1f
    csrw stvec, t0

    # set satp with early_pgtbl
    la t0, early_pgtbl
    sub t0, t0, t1
    srl t0, t0, 12
    li t1, 0x8000000000000000
    or t0, t0, t1
    csrw satp, t0


    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret

1:
    li t0, 0xffffffdf80000000
    csrr t1, sepc
```

```
    add t1, t1, t0
    csrw sepc, t1
    ret
```

运行结果如下，可以正常编译运行：

```
Boot HART MHPM Count        : 0
Boot HART MIDELEG           : 0x0000000000000222
Boot HART MEDELEG           : 0x000000000000b109
...setup_vm done!
...mm_init done!
...setup_vm_final done!
...proc_init done!
2022 Hello RISC-V
sstatus = 8000000000006102
sscratch = 0
[INTERRUPT] S mode timer interrupt!

switch to [PID = 8 COUNTER = 1 PRIORITY = 25]
[PID = 8] is running. auto_inc_local_var = 1
[INTERRUPT] S mode timer interrupt!

switch to [PID = 17 COUNTER = 1 PRIORITY = 71]
[PID = 17] is running. auto_inc_local_var = 1
[INTERRUPT] S mode timer interrupt!

switch to [PID = 18 COUNTER = 1 PRIORITY = 64]
[PID = 18] is running. auto_inc_local_var = 1
```