lab4

4.2 开启虚拟内存映射

4.2.1 setup_vm 的实现

setup_vm的实现需要实现两个函数，一个是setup_vm，一个是relocate。

对于setup_vm，我们只要实现页表的映射即可。其中 `PHY_START >> 30` 是去掉后面30位，只留下PPN2，之后再把PPN2放到页表对应的位置上，再加上表示权限的flag位即可。

```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
    */

    // here the flag 10 bits
    unsigned long flag = 0b1111;
    // here the early_pgtbl is physical address
    // PA = VA
    early_pgtbl[PHY_START >> 30] = (PHY_START >> 30 << 28) + flag;
    // you need to flash high bit
    // PA + PA2VA_OFFSET = VA
    early_pgtbl[VM_START << 25 >> 25 >> 30] = (PHY_START >> 30 << 28) + flag;
    // printk("%lx, early_pgtbl[%lx]: %lx\n\n", early_pgtbl, VM_START << 25 >> 25 >> 30,
    early_pgtbl[VM_START << 25 >> 25 >> 30]);

}
```

需要注意的是，因为要调用setup_vm，所以要提前设置好sp的值，又因为当我们还未调用relocate的时候，虚拟内存是无法使用的，因此我们要将sp置为boot_stack_up并将其减去 `PA2VA_OFFSET` ，才算是使用物理内存。

```
# set sp
la sp, boot_stack_top
li t0, 0xffffffdf80000000
sub sp, sp, t0
add s0, sp, x0

# init virtual memory
call setup_vm
call relocate
```

之后就是relocate的实现了，注意，由于之前的sp已经被初始化成物理地址，因此在这里要加上 `PA2VA_OFFSET` 才能转到虚拟地址。之后是设置satp，前4位设置MODE位，之后再把early_pgtbl的物理页号存入后44位。

```
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

    # set satp with early_pgtbl
    # set MODE
    addi t2, x0, 8
    slli t2, t2, 60
    # set PPN
    la t1, early_pgtbl
    sub t1, t1, t0
    srli t1, t1, 12

    add t1, t1, t2
    csrw satp, t1

    # flush tlb
    sfence.vma zero, zero

    # flush icache
    fence.i

    ret

    .section .bss.stack
    .globl boot_stack

boot_stack:
    .space 0x1000 # <-- change to your stack size
```

这样，我们的setup_vm就完成了。跑一下，我们可以得到下面的结果。可以看到，程序在虚拟地址上正常运行。

```
SET [PID = 30 COUNTER = 8]
SET [PID = 31 COUNTER = 8]
[PID = 12] is running. thread space begin at 0xffffffe007ff3000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 28] is running. thread space begin at 0xffffffe007fe3000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. thread space begin at 0xffffffe007ffe000
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. thread space begin at 0xffffffe007ffe000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 2] is running. thread space begin at 0xffffffe007ffd000
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 2] is running. thread space begin at 0xffffffe007ffd000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 9] is running. thread space begin at 0xffffffe007ff6000
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 9] is running. thread space begin at 0xffffffe007ff6000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 14] is running. thread space begin at 0xffffffe007ff1000
current->counter = 2
```

4.2.2 setup_vm_final 的实现

之后是setup_vm_final的实现。其做的事跟setup_vm实际上差不多，只不过是多映射了一些以及变成了三级页表。

首先，修改mm_init中的值使其malloc对应虚拟地址上的页。这里要注意物理内存只有128M，所以range也只有128M。

```
void mm_init(void) {
    // here the all physical memory is 128M        You, now • Unc
    kfreerange(_ekernel, (char *)(VM_START+0x8000000));
    printk("...mm_init done!\n");
}
```

之后是setup_vm_final的实现。首先，我们将需要的有关kernel段的位置引入进来，注意类型一定要定义为char *。

```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[];
extern char _etext[];
extern char _srodata[];
extern char _erodata[];
extern char _sdata[];
```

之后是映射的实现。由于我们要映射128M的地址，所以得用循环。text rodata data段的权限各不相同。对于映射的pgtbl，由于先前实现了等值映射，我直接将swapper_pg_dir的物理地址传了进去。

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

    // No OpenSBI mapping required

    uint64 va;
    uint64 *pa_swapper_pg_dir = (uint64 *)((uint64)swapper_pg_dir - PA2VA_OFFSET);
    printk("_stext: 0x%lx\n", _stext);
    printk("_etext: 0x%lx\n", _etext);
    printk("_srodata: 0x%lx\n", _srodata);
    printk("_erodata: 0x%lx\n", _erodata);
    printk("_sdata: 0x%lx\n", _sdata);
    printk("_end: 0x%lx\n", VM_START + PHY_SIZE);
    printk("swapper_pg_dir: 0x%lx\n", swapper_pg_dir);
    printk("pa_swapper_pg_dir: 0x%lx\n\n\n", pa_swapper_pg_dir);

    // mapping kernel text X|-|R|V
    for(va = _stext; va < _etext; va += PGSIZE){
        // printk("va: %lx\n", va);
        create_mapping(pa_swapper_pg_dir, va, va - PA2VA_OFFSET, PGSIZE, 0b1011);
    }
    // create_mapping(swapper_pg_dir, _stext,  _stext - PA2VA_OFFSET, PGSIZE, 0b1011);
    // while (1);

    // mapping kernel rodata -|-|R|V
    for(va = _srodata; va < _erodata; va += PGSIZE){
        create_mapping(pa_swapper_pg_dir, va, va - PA2VA_OFFSET, PGSIZE, 0b0011);
    }
    // create_mapping(pa_swapper_pg_dir, _srodata,  _srodata - PA2VA_OFFSET, PGSIZE,
    0b0011);

    // mapping other memory -|W|R|V
    for(va = _sdata; va < VM_START + PHY_SIZE; va += PGSIZE){
        // printk("va: %lx\n", va);
        create_mapping(pa_swapper_pg_dir, va, va - PA2VA_OFFSET, PGSIZE, 0b0111);
    }
```

注意权限的设置一定要正确，不然qemu就会跳到trap里，scause为0xf。也就是 `Store/AMO page fault` - 当试图向一个没有写权限的页写入时产生。

```
Breakpoint 1, setup_vm_final () at vm.c:50
50          memset(swapper_pg_dir, 0x0, PGSIZE);
(gdb) i r satp
satp           0x8000000000080206       -9223372036854251002
(gdb) i r stvec
stvec          0x80200000       2149580800
(gdb) b * 0x80200000
Breakpoint 2 at 0x80200000
(gdb) c
Continuing.

Breakpoint 2, 0x0000000080200000 in ?? ()
(gdb) i r sstatus
sstatus        0x8000000000006100       -9223372036854750976
(gdb) i r scause
scause         0xf      15
(gdb) 
```

这里W置1但是R置0了，导致错误。

```c
    printk("_end: 0x%lx\n\n\n", VM_START + PHY_SIZE);
    printk("swapper_pg_dir: 0x%lx\n", swapper_pg_dir);

    for(va = _stext; va < _etext; va += PGSIZE){
        // printk("va: %lx\n", va);
        create_mapping(swapper_pg_dir, va, va - PA2VA_OFFSET, PGSIZE, 0b1011);
    }
    // create_mapping(swapper_pg_dir, _stext,  _stext - PA2VA_OFFSET, PGSIZE, 0b1011);

    // mapping kernel rodata -|-|R|V
    create_mapping(swapper_pg_dir, _srodata,  _srodata - PA2VA_OFFSET, PGSIZE, 0b0011);

    // mapping other memory -|W|R|V
    for(va = _sdata; va < VM_START + PHY_SIZE; va += PGSIZE){
        // printk("va: %lx\n", va);
        create_mapping(swapper_pg_dir, va, va - PA2VA_OFFSET, PGSIZE, 0b0101);
    }

    // set satp with swapper_pg_dir
```

之后是satp的设置。注意存的一定是物理地址，也可以使用之前实现的csr_write。

```c
    // set satp with swapper_pg_dir
    uint64 val = ((uint64)8 << 60) + ((uint64)(pa_swapper_pg_dir) >> 12);
    csr_write(satp, val);
    printk("satp: %lx\n", csr_read(satp));

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

    // flush icache
    asm volatile("fence.i");
    printk("...set_up_final done!\n");
    return;
}
```

之后是create_mapping。其传入根页表的地址，并传入要映射的va,pa,sz被固定为PG_SIZE，perm则是权限。 我们首先将每集页表对应的PAGE NUMBER表示出来，再调用get_next_pgtbl_base，返回下级页表的基地址。最后把对应的物理页号放在最后的页表项中。

```c
/* 创建多级页表映射关系 */
// 这是一个只支持sz = PGSIZE的
create_mapping(uint64 *pgtbl2, uint64 va, uint64 pa, uint64 sz, int 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
    perm 为映射的读写权限

    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
    */
    // for sz < PGSIZE
    uint64 VPN2 = va << 25 >> 25 >> 30;
    uint64 VPN1 = va << 34 >> 34 >> 21;
    uint64 VPN0 = va << 43 >> 43 >> 12;
    uint64 VOFF = va << 52 >> 52;
    uint64 PPN = pa >> 12;
    uint64 POFF = pa << 52 >> 52;
    if (sz != PGSIZE){
        printk("ERROR: sz != PGSIZE\n");
    }
    if (VOFF != POFF){
        printk("ERROR: VOFF != POFF\n");
    }

    uint64 *pgtbl1 = get_next_pgtbl_base(pgtbl2, VPN2);
    // printk("%lx, pgtbl2[%lx]: %lx, pn = %lx\n", pgtbl2, VPN2, pgtbl2[VPN2], pgtbl2
    [VPN2] >> 10 << 12);
    uint64 *pgtbl0 = get_next_pgtbl_base(pgtbl1, VPN1);
    // printk("%lx, pgtbl1[%lx]: %lx, pn = %lx\n", pgtbl1, VPN1, pgtbl1[VPN1], pgtbl1
    [VPN1] >> 10 << 12);
    // map PPN
    pgtbl0[VPN0] = (PPN << 10) + perm;
    // printk("pgtbl0 = %lx, PPN = %lx\n", pgtbl0, pgtbl0[VPN0]>>10<<12);
```

一定要注意的是，页表项R, W, X如果三位都为0，表示是一个指向下一页表的指针。这里我忘记了，导致了之后页表查询没什么反应。

```
0000000080207000, pgtbl2[0000000000000180]: 3ffffff801fffc0b, pn = ffffffe007fff000
ffffffe007fff000, pgtbl1[0000000000000001]: 3ffffff801fff80b, pn = ffffffe007ffe000
ffffffe007ffe000, pgtbl0[00000000000000da]: 00000000200b6805, pn = 00000000802da000
0000000080207000, pgtbl2[0000000000000180]: 3ffffff801fffc0b, pn = ffffffe007fff000
ffffffe007fff000, pgtbl1[0000000000000001]: 3ffffff801fff80b, pn = ffffffe007ffe000
ffffffe007ffe000, pgtbl0[00000000000000db]: 00000000200b6c05, pn = 00000000802db000
```

最后是get_next_pgtbl_base的实现，我们传入上一级的页表基地址和虚拟页号作为OFFSET，之后check entry的Valid bit，如果是1就返回对应的entry中下一级的页表基地址。如果是0则要kalloc一个新页表，注意一定要存入对应的物理页号，再把这个next_pgtbl传回去。

```c
// it just get pgtbl 2/1, not 0
// + is super than <<
uint64 *get_next_pgtbl_base(uint64 *pgtbl, uint64 VPN){
    uint64 next_pgtbl;
    // V = 1
    // no priority control
    if(pgtbl[VPN] % 2){
        next_pgtbl = pgtbl[VPN] >> 10 << 12;
    }else{
        next_pgtbl = kalloc() - PA2VA_OFFSET;
        // pgn = address >> 12
        pgtbl[VPN] = (next_pgtbl >> 12 << 10) + 1;
    }
    return (uint64 *)next_pgtbl;
}
```

这样，我们的setup_vm_final就实现好了。最后的实现效果如图所示，可以看到与前面是一样的。

```
SET [PID = 26 COUNTER = 6]
SET [PID = 27 COUNTER = 10]
SET [PID = 28 COUNTER = 1]
SET [PID = 29 COUNTER = 3]
SET [PID = 30 COUNTER = 8]
SET [PID = 31 COUNTER = 8]
[PID = 12] is running. thread space begin at 0xffffffe007fb3000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 28] is running. thread space begin at 0xffffffe007fa3000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. thread space begin at 0xffffffe007fbe000
current->counter = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. thread space begin at 0xffffffe007fbe000
current->counter = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 2] is running. thread space begin at 0xffffffe007fbd000
current->counter = 2
```

思考题

1. 验证 .text, .rodata 段的属性是否成功设置，给出截图。

可以看对应的table entry后三位，1011是.text 0011是.rodata 0111是其他memory 属性设置是正确的。

```
0000000080207000, pgtbl2[0000000000000180]: 0000000021fffc01, pn = 0000000087fff000
0000000087fff000, pgtbl1[0000000000000001]: 0000000021fff801, pn = 0000000087ffe000
0000000087ffe000, pgtbl0[0000000000000000]: 000000002008000b, pn = 0000000080200000

0000000080207000, pgtbl2[0000000000000180]: 0000000021fffc01, pn = 0000000087fff000
0000000087fff000, pgtbl1[0000000000000001]: 0000000021fff801, pn = 0000000087ffe000
0000000087ffe000, pgtbl0[0000000000000001]: 000000002008040b, pn = 0000000080201000

0000000080207000, pgtbl2[0000000000000180]: 0000000021fffc01, pn = 0000000087fff000
0000000087fff000, pgtbl1[0000000000000001]: 0000000021fff801, pn = 0000000087ffe000
0000000087ffe000, pgtbl0[0000000000000002]: 0000000020080803, pn = 0000000080202000

0000000080207000, pgtbl2[0000000000000180]: 0000000021fffc01, pn = 0000000087fff000
0000000087fff000, pgtbl1[0000000000000001]: 0000000021fff801, pn = 0000000087ffe000
0000000087ffe000, pgtbl0[0000000000000003]: 0000000020080c07, pn = 0000000080203000
```

2. 为什么我们在 setup_vm 中需要做等值映射?

因为这能让我们在虚拟地址上和虚拟地址对应的物理地址上都有kernel代码的映射，为后面setup_vm_final能通过虚拟地址访问物理地址做准备，
而且如果没有等值映射，pc还是跑在低地址，PA!=VA就会导致异常，不能平滑的切换虚拟地址和物理地址。

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。

我们可以看 `arch/riscv/kernel/head.S` 中的 `clear_bss_done` 段（301行），这是_start_kernel中的一段，其中设置了sp之后，其 `call setup_vm` 设置初始的页表之后，将early_pg_dir作为参数传入了relocate之中。

```
        /* Initialize page tables and relocate to virtual addresses */
        la tp, init_task
        la sp, init_thread_union + THREAD_SIZE
        XIP_FIXUP_OFFSET sp
#ifdef CONFIG_BUILTIN_DTB
        la a0, __dtb_start
        XIP_FIXUP_OFFSET a0
#else
        mv a0, s1
#endif /* CONFIG_BUILTIN_DTB */
        call setup_vm
#ifdef CONFIG_MMU
        la a0, early_pg_dir
        XIP_FIXUP_OFFSET a0
        call relocate_enable_mmu
#endif /* CONFIG_MMU */
```

之后就是relocate的部分。其中第一部分就是加上虚拟地址和物理地址的偏移量。之后就是设置stvec为1f和计算early_pg_dir的虚拟地址。但其先不写入satp，而是先写入trampoline_pg_dir，从而开启mmu。

开启mmu之后直接触发异常进入 `1f` ，其中设置了stvec是 `.Lsecondary_park` ，这里就是个死循环，用于debug。继续设置satp为传入的early_pg_table，如果没有触发异常，说明页表设置成功，这样程序就在虚拟地址上跑了。最后返回_start_kernel。

```
relocate_enable_mmu:
        /* Relocate return address */
        la a1, kernel_map
        XIP_FIXUP_OFFSET a1
        REG_L a1, KERNEL_MAP_VIRT_ADDR(a1)
        la a2, _start
        sub a1, a1, a2
        add ra, ra, a1

        /* Point stvec to virtual address of intruction after satp write */
        la a2, 1f
        add a2, a2, a1
        csrw CSR_TVEC, a2

        /* Compute satp for kernel page tables, but don't load it yet */
        srl a2, a0, PAGE_SHIFT
        la a1, satp_mode
        REG_L a1, 0(a1)
        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
        XIP_FIXUP_OFFSET a0
        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_TVEC, 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 guaranteed to work
         * because that first superpage is translated the same way.
         */
```

```
        csrw CSR_SATP, a2
        sfence.vma

        ret
```