实验三-RV64 虚拟内存管理

姓名:王晶晶 学号:3200104880 学院:计算机科学与技术学院 课程名称：计算机系统Ⅲ

实验时间：2022.4.28 实验地点：紫金港东4-509 指导老师：周亚金

一、实验目的和要求

- 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。

二、实验原理

2.1 虚拟内存

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为按需分页（demand paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为页换入换出（page swap in/out）。这种内存管理技术给了程序员更大的内存“空间”，从而可以让更多的程序在内存中并发运行。

2.2 MMU

MMU（Memory Management Unit）是一种负责处理中央处理器（CPU）的内存访问请求的计算机硬件。它的功能包括虚拟地址到物理地址的转换（即虚拟内存管理）、内存保护、中央处理器高速缓存的控制。MMU位于处理器内核和连接高速缓存以及物理存储器的

总线之间。如果处理器没有MMU，CPU内部执行单元产生的内存地址信号将直接通过地址总线发送到芯片引脚，被内存芯片接收，这就是物理地址。如果MMU存在且启用，CPU执行单元产生的地址信号在发送到内存芯片之前将被MMU截获，这个地址信号称为虚拟地址，MMU会负责把VA翻译成相应的物理地址，然后发到内存芯片地址引脚上。

2.3 分页机制

分页机制的基本思想是将程序的虚拟地址空间划分为连续的，等长的虚拟页。虚拟页和物理页的页长固定且相等（一般情况下为4KB），从而操作系统可以方便的为每个程序构造页表，即虚拟页到物理页的映射关系。

逻辑上，该机制下的虚拟地址有两个部分组成：1.虚拟页号；2.页内偏移。在具体的翻译过程中，MMU首先解析得到虚拟地址中的虚拟页号，并通过虚拟页号查找到对应的物理页，用该物理页的起始地址加上页内偏移得到最终的物理地址。

本次实验中，使用RISCV Sv39 三级页表分页机制。 Sv39使用4KB大的基页，页表项的大小是8个字节，为了保证页表大小和页面大小一致，树的基数相应地降到2^9，树也变为三层。 Sv39的 512 GB地址空间（虚拟地址）划分为2^9个 1GB大小的吉页 。每个基页被进一步划分为2^9个2MB大小的巨页。每个巨页再进一步分为2^9个4KB大小的基页。

2.4 RV64虚拟内存管理

在本次实验中，为了让线程之间的运行空间相互独立，引入虚拟地址空间。为了开启虚拟映射，主要有以下几个步骤。

- 初始化根页表，对内核代码进行等值映射
- 改变映射模式为Sv39，分别定位satp的MODE,ASID,PPN字段并进行写入
- 初始化地址空间
- 初始化三级页表，根据text段、rodata段、data等各段的访问属性及地址VPN[2]-VPN[0]初始化三级页表映射
- 创建线程并分配运行空间

三、实验代码实现

3.1 初始化根页表，对内核代码进行等值映射

```
//VA 0x80000000   index=10
early_pgtbl[2]=(unsigned long)(((0x80000000>>12)<<10)|0x000000000000000F);

//VA 0xffffffe00000000  e-1110 index=1 10000000=0x180
early_pgtbl[0x180]=(unsigned long)(((0x80000000>>12)<<10)|0x000000000000000F);
```

3.2 改变映射模式为Sv39，设置satp寄存器

```
# set satp with early_pgtbl
    la t1,early_pgtbl
    srli t1,t1,12
    li t2,0x8000000000000000
    or t1,t1,t2
    csrw satp,t1
```

3.3 初始化三级页表

```
if((*first)&0x1)
                second=(unsigned long*)((unsigned long)(((*first)>>10)<<12)+PA2VA_OFF
            //if frist level page is not valid, allocate one page for it and fill the
             else{
                second=(uint64*)kalloc();
                memset(second,0,PGSIZE);
                *first=(unsigned long)((*first)&0xffc0000000000000)|
                (((((unsigned long)second-PA2VA_OFFSET)>>12)<<10)|(0x1);
             }
```

四、实验结果分析

在启动OpenSBI后，kernel会先初始化内存，输出"mm_init done!"表示对于从kernel结束地址开始的虚拟地址都已经被初始化。

完成后再开启虚拟映射并初始化页表，最后初始化线程。输出"proc_init done!"开始执行线程之间的切换。

```
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
...mm_init done!
...proc_init done!
 Hello RISC-V
SET [PID = 1 PRIORITY = 1 COUNTER = 2]
SET [PID = 2 PRIORITY = 4 COUNTER = 7]
SET [PID = 3 PRIORITY = 10 COUNTER = 5]
SET [PID = 4 PRIORITY = 4 COUNTER = 5]
SET [PID = 5 PRIORITY = 10 COUNTER = 10]
SET [PID = 6 PRIORITY = 10 COUNTER = 9]
SET [PID = 7 PRIORITY = 5 COUNTER = 10]
SET [PID = 8 PRIORITY = 2 COUNTER = 2]
SET [PID = 9 PRIORITY = 9 COUNTER = 6]
SET [PID = 10 PRIORITY = 4 COUNTER = 3]
SET [PID = 11 PRIORITY = 4 COUNTER = 1]
SET [PID = 12 PRIORITY = 10 COUNTER = 10]
```

可以看到每个线程都被分配到一个独立的虚拟地址空间，输出的值均以 `ffffffe` 开头，说明进程被分配到的虚拟空间地址。

当一个进程的时间片用完，发生进程调度切换到下一个进程时，会输出写一个进程分配的虚拟地址。

```
SET [PID = 30 PRIORITY = 9 COUNTER = 8]
SET [PID = 31 PRIORITY = 4 COUNTER = 8]
SET [PID = 32 PRIORITY = 2 COUNTER = 3]
switch to [PID = 11 PRIORITY = 4 COUNTER = 1]
[PID = 11] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fb4000
switch to [PID = 27 PRIORITY = 1 COUNTER = 1]
[PID = 27] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fa4000
switch to [PID = 1 PRIORITY = 1 COUNTER = 2]
[PID = 1] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fbe000
[PID = 1] is running. auto_inc_local_var = 2 thread space begin at ffffffe007fbe000
switch to [PID = 8 PRIORITY = 2 COUNTER = 2]
[PID = 8] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fb7000
[PID = 8] is running. auto_inc_local_var = 2 thread space begin at ffffffe007fb7000
switch to [PID = 13 PRIORITY = 5 COUNTER = 2]
[PID = 13] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fb2000
[PID = 13] is running. auto_inc_local_var = 2 thread space begin at ffffffe007fb2000
switch to [PID = 10 PRIORITY = 4 COUNTER = 3]
[PID = 10] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fb5000
[PID = 10] is running. auto_inc_local_var = 2 thread space begin at ffffffe007fb5000
[PID = 10] is running. auto_inc_local_var = 3 thread space begin at ffffffe007fb5000
switch to [PID = 28 PRIORITY = 3 COUNTER = 3]
[PID = 28] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fa3000
[PID = 28] is running. auto_inc_local_var = 2 thread space begin at ffffffe007fa3000
[PID = 28] is running. auto_inc_local_var = 3 thread space begin at ffffffe007fa3000
switch to [PID = 32 PRIORITY = 2 COUNTER = 3]
[PID = 32] is running. auto_inc_local_var = 1 thread space begin at ffffffe007f9f000
[PID = 32] is running. auto_inc_local_var = 2 thread space begin at ffffffe007f9f000
[PID = 32] is running. auto_inc_local_var = 3 thread space begin at ffffffe007f9f000
switch to [PID = 14 PRIORITY = 10 COUNTER = 4]
[PID = 14] is running. auto_inc_local_var = 1 thread space begin at ffffffe007fb1000
```

五、实验中遇到的问题及解决方法

- 一开始gdb的时候不对劲
  根据System.map,一开始打断点要打在物理地址的_start 0x80200000处，在设置satp开启Sv30虚拟映射之后，再根据虚拟地址空间打断点调试。
- 程序一直不断重复输出"mm_init done!"
  是三级页表设置的问题

```
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
...mm_init done!
```

六、思考题与心得体会

- 验证 .text, .rodata 段的属性是否成功设置，给出截图。
  在代码中添加 `printk()` 打印出具体信息。

```
// mapping kernel text X|-|R|V
unsigned long text_sz=(unsigned long)&_srodata-(unsigned long)&_stext;
create_mapping(tmp,va,pa,text_sz,11);
printk("text section: %lx~%lx\n",va,va+text_sz);
// mapping kernel rodata -|-|R|V
va=va+text_sz;
pa=pa+text_sz;
unsigned long rodata_sz=(unsigned long)&_sdata-(unsigned long)&_srodata;
create_mapping(tmp,va,pa,rodata_sz,3);
printk("rodata section: %lx~%lx\n",va,va+rodata_sz);
```

```
            //third level fill it with corresponding pa
            third=&third[((va+i)>>12)&0x1FF];
            *third=(unsigned long)(((*third)&0xffc0000000000000)|((
|(unsigned long)perm));
        }
        printk("third level pa: %lx\n",*third);
        return;
```

结果如下。

```
...mm_init done!
third level pa: 000000002008040b
text section: ffffffe000200000~ffffffe000202000
third level pa: 0000000020080803
rodata section: ffffffe000202000~ffffffe000203000
```

图中可以看到， `.text` 段三级页表最后一级的值为000000002008040b。最后四位为1011,表示X|-|R|V。

`.rodata` 段三级页表最后一级的值为0000000020080803。最后四位为0011,表示-|-|R|V。

- 为什么我们在 setup_vm 中需要做等值映射?

  0x80000000开始的地址存放着内核代码和OpenSBI,这部分的代码对于不同进程来说是完全相同的。在进程切换时，为了保持内核态地址是完全相同的，即所有进程的内核地址映射完全一致，需要对这部分地址空间进行等值映射。

  另一方面保证了，分页后执行尚未执行完的代码时还是处在正确的kernel代码段。

  在RISC-V 64中，机器模式使用物理地址，而监管者模式、用户模式使用虚拟地址，为了保证模式切换地址映射的一致性，只要在机器模式下确定的地址，且需要在监管者模式下使用的，都需要等值映射。

- 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。

  当执行到 `csrw satp` 处之后，由于开启了虚拟地址映射，原来处在物理地址的代码段将会发生 `page fault` ，linux通过捕获此类异常，并设置中断向量入口地址为物理地址加上偏移量后对应的虚拟地址，再将控制流交还给程序。从而不用在setup_vm中再做等值映射。

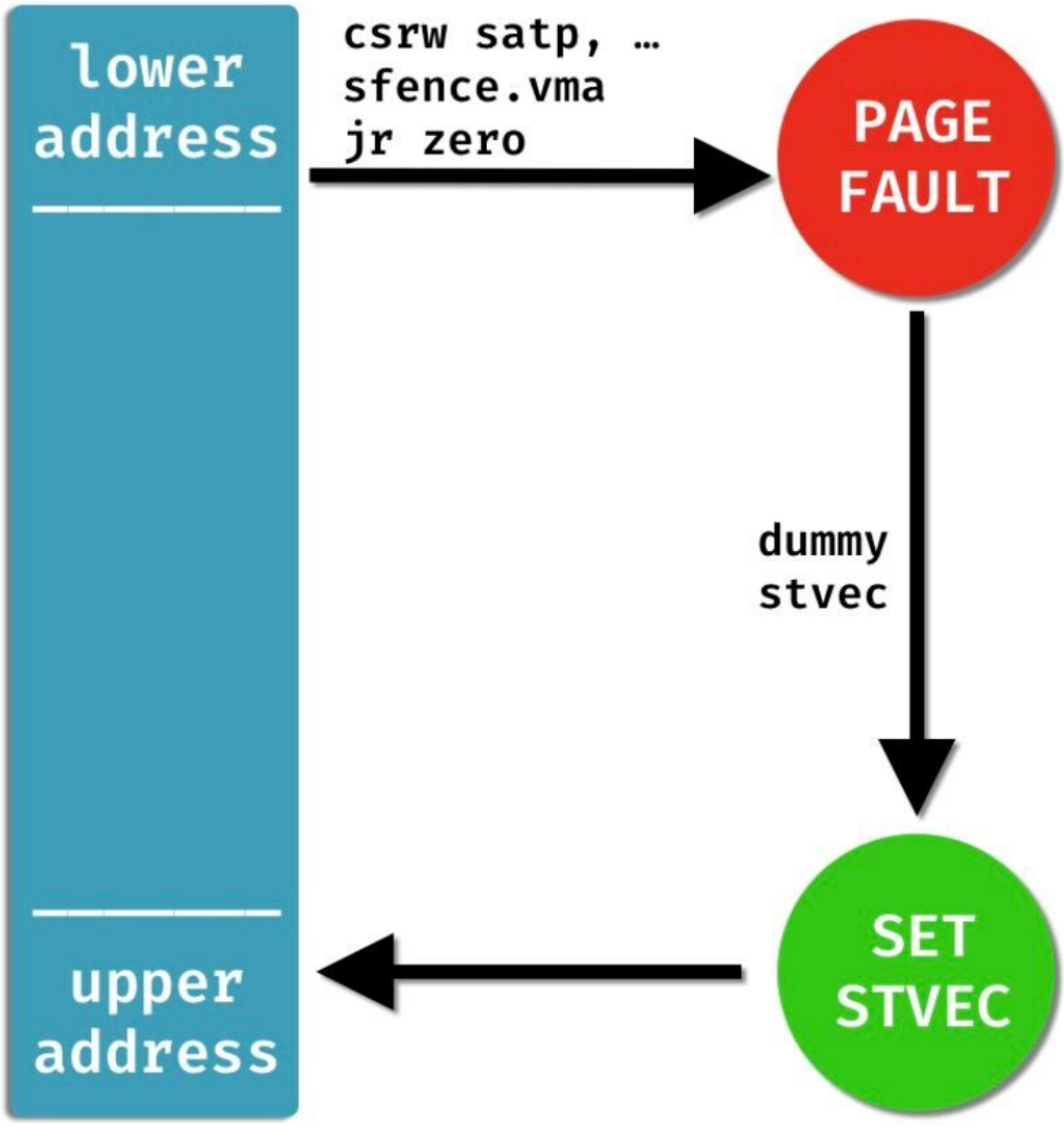

图 1: Jump to Virtual Memory