Lab10 - Page Table in RISC-V & User Space

0. 前言

实验概述

本实验将学习 RISC-V 架构下的页表映射机制以及用户态空间的管理代码。

实验代码

本节实验课没有新代码，可以继续使用lab9的代码。

1. RISC-V下的页表

在现代计算机系统中，页表是实现虚拟内存管理的核心机制。它通过建立虚拟地址到物理地址的映射关系，为每个进程提供独立的地址空间，同时隔离用户空间与内核空间，从而实现内存隔离和保护。当程序访问虚拟地址时，内存管理单元（MMU）会自动查询页表，将其转换为对应的物理地址。

RISC-V 使用页表翻译机制实现虚拟地址到物理地址的映射，该机制由以下组件协同工作：

- **satp 寄存器**：存储根页表的物理地址，并配置地址转换模式；
- **多级页表结构**：例如 Sv48 采用四级页表，逐级完成虚拟地址到物理地址的翻译；
- **页表项（PTE）**：包含物理页号（PPN）以及控制标志位，如 R/W/X 权限位、有效位 V、用户位 U 等；
- **快表（TLB）**：缓存近期使用的地址映射，提升地址转换性能。

1.1 satp 寄存器

RISC-V 使用 `satp` 寄存器存储页表基地址，作为多级页表映射的起始点。除了根页表基地址外，该寄存器还包含页表模式等配置信息。`satp` 的低 44 位为 **PPN（Physical Page Number）**，其值乘以页大小（4 KB = 2^12）即为根页表的物理地址。

`satp` 寄存器的结构及其支持的页表模式如下：

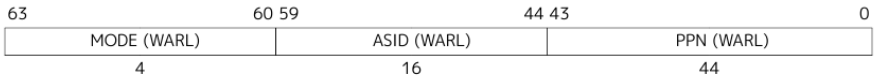


Figure 57. Supervisor address translation and protection (`satp`) register when SXLEN=64, for MODE values Bare, Sv39, Sv48, and Sv57.

Table 24. Encoding of `satp` MODE field.

SXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section 11.3).
SXLEN=64		
Value	Name	Description
0	Bare	No translation or protection.
1-7	-	*Reserved for standard use*
8	Sv39	Page-based 39-bit virtual addressing (see Section 11.4).
9	Sv48	Page-based 48-bit virtual addressing (see Section 11.5).
10	Sv57	Page-based 57-bit virtual addressing (see Section 11.6).
11	Sv64	*Reserved for page-based 64-bit virtual addressing.*
12-13	-	*Reserved for standard use*
14-15	-	*Designated for custom use*

1.2 Sv48映射

目前 OSTD 0.16.1 版本仅支持 RISC-V 的 Sv48 页表映射机制。与 Sv39 三级页表相比，Sv48 增加了一级映射（从 39 位扩展至 48 位虚拟地址），其余结构基本保持一致。

物理地址与虚拟地址定义

在 Sv48 中，物理地址为 56 位，虚拟地址为 48 位。虚拟地址虽然占用 64 位，但仅低 48 位有效，且第 48 至 63 位的值必须与第 47 位相同，否则视为非法地址。结构如下：

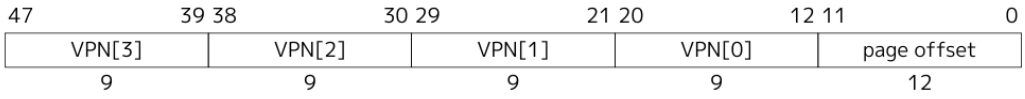


Figure 64. Sv48 virtual address.

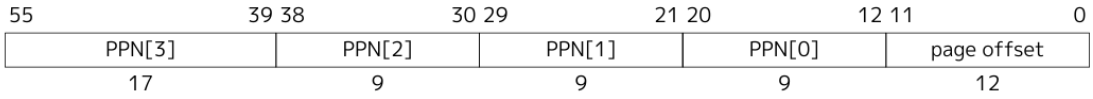


Figure 65. Sv48 physical address.

页表项

Sv48的页表项定义如下：

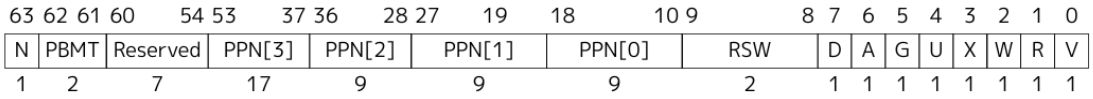


Figure 66. Sv48 page table entry.

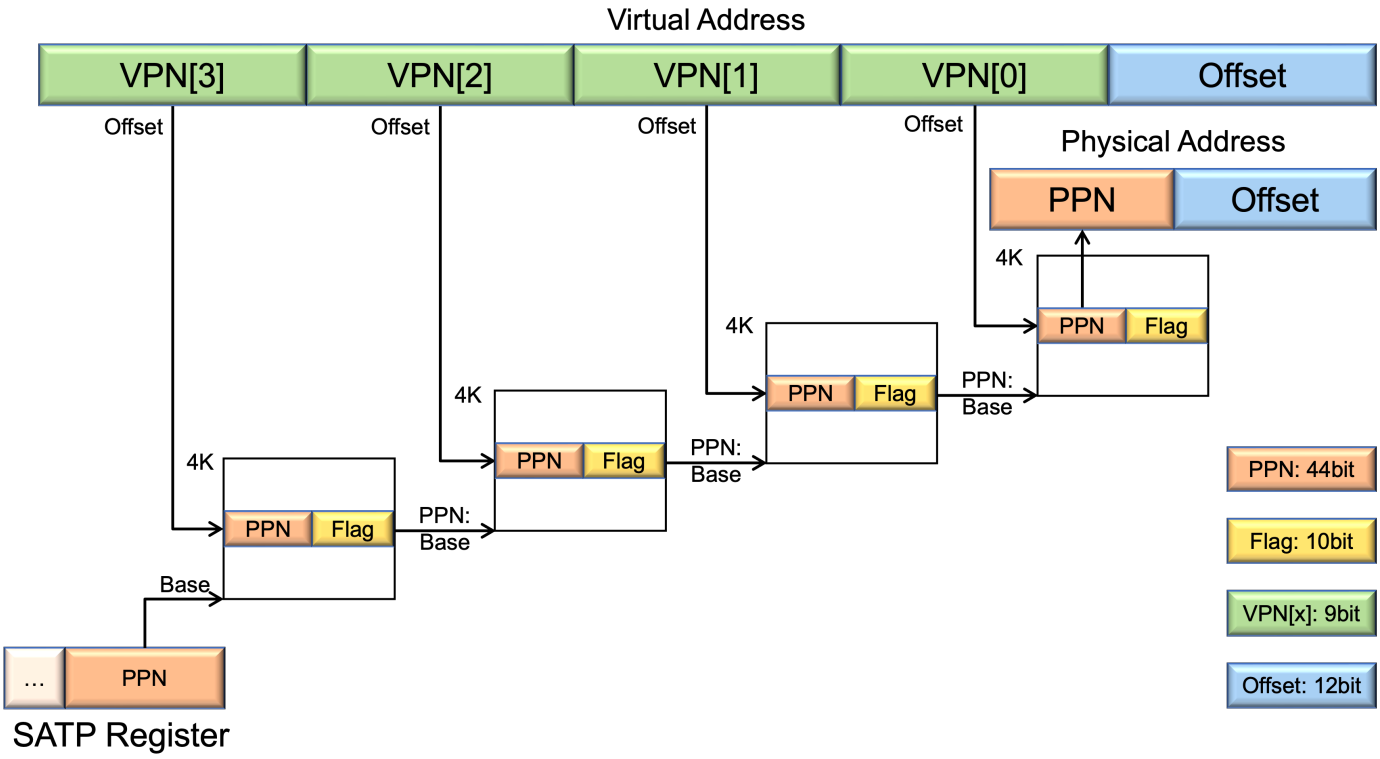
其中，位 [63:54] 为保留位，可忽略；位 [53:10] 为 44 位物理页号（PPN）；最低 10 位为标志位，其中 [9:8] 为用户自定义标志，其余为架构定义标志，具体含义如下：

- **V（Valid）**：表示该映射是否有效；
- **R（Read）/ W（Write）/ X（eXecute）**：控制对应虚拟页是否可读、可写、可执行；
- **U（User）**：表示用户态（U 模式）下是否可以访问该虚拟页；
- **G（Global）**：表示该页表项为全局项，与 TLB 机制相关；
- **A（Accessed）**：若为 1，表示自上次清零后，有虚拟地址通过该页表项进行读、写或取指操作；
- **D（Dirty）**：若为 1，表示自上次清零后，有虚拟地址通过该页表项进行写入操作。

RWX 标志位的不同组合具有不同含义：

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

当 RWX 全为 0 时，该页表项指向下一级页表；若任一 RWX 位非零，则该页表项为叶子节点，其 PPN 即为最终物理页的基地址。Sv48整体映射结构如下：



如果在某一级（如第二级）页表中设置 RWX 为非零，则会形成“大页”映射。此时，CPU 会将后续未翻译的虚拟地址位视为偏移量。例如，合并 VPN[0] 和页偏移，可将偏移扩展至 12（offset）+ 9（VPN[0]） 位，实现 2 MB 的大页映射。

OSTD中的RISC-V页表

OSTD 在 `arch/riscv/mm/mod.rs` 中定义了页表相关结构，包括页表项、flag、与页表常数，其中页表常数包含基础页大小（4096 bytes）、页表级数（4 level）、地址宽度（48 bits）、虚拟地址符号扩展、最高可用于映射物理地址的级数（4 level，该常数是为了支持大页）、页表项大小（64 bits）：

```
#[derive(Clone, Debug, Default)]
pub struct PagingConsts {}

impl PagingConstsTrait for PagingConsts {
    const BASE_PAGE_SIZE: usize = 4096;
    const NR_LEVELS: PagingLevel = 4;
    const ADDRESS_WIDTH: usize = 48;
    const VA_SIGN_EXT: bool = true;
    const HIGHEST_TRANSLATION_LEVEL: PagingLevel = 4;
    const PTE_SIZE: usize = size_of::<PageTableEntry>();
}

bitflags::bitflags! {
    #[derive(Pod)]
    #[repr(C)]
    pub struct PageTableFlags: usize {
        const VALID =           1 << 0; // 0000_0001
        const READABLE =        1 << 1; // 0000_0010
        const WRITABLE =        1 << 2; // 0000_0100
        const EXECUTABLE =      1 << 3; // 0000_1000
        const USER =            1 << 4; // 0001_0000
```

```
        ...
    }
}

#[derive(Clone, Copy, Pod, Default)]
#[repr(C)]
pub struct PageTableEntry(usize);

impl PageTableEntryTrait for PageTableEntry {
    ...
}
```

此外，OSTD 还定义了跨平台的 `PageProperty` 结构：

```
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct PageProperty {
    pub flags: PageFlags,
    pub cache: CachePolicy,
    pub(crate) priv_flags: PrivilegedPageFlags,
}
```

> **Bitflags**
>
> `bitflags` 是 Rust 生态中处理位标志的实用库。在操作系统开发中，常用位标志来表示功能开关，RWX 权限位就是典型的位标志应用。

1.3 快表

物理内存的访问速度远低于 CPU 处理速度，一次内存访问可能需数百个时钟周期，形成“冯诺依曼瓶颈”。若每次地址转换都遍历四级页表，需访问 4 次内存获取物理地址，再访问 1 次以获取数据，将严重影响性能。

实践中，虚拟地址访问具有局部性特征：

- **时间局部性**：近期被访问的地址很可能在不久后再次被访问；
- **空间局部性**：某个地址被访问后，其邻近地址也可能很快被访问。

为此，CPU 内部使用**快表（TLB, Translation Lookaside Buffer）**缓存近期使用的虚拟页号到物理页号的映射。利用局部性，TLB 能在多数情况下直接提供映射结果，避免多次内存访问。

需要注意的是，当修改 `satp` 寄存器以切换页表时，TLB 中缓存的映射可能失效。此时需使用 `sfence.vma` 指令刷新整个 TLB。同样，手动修改页表项后，TLB 不会自动更新，也需使用 `sfence.vma` 指令刷新映射。该指令可不带参数刷新整个 TLB，也可指定虚拟地址仅刷新该地址的映射项。

2. OSTD中的页表操作

一般而言，一个操作系统需要提供若干页表操作的接口，包括但不限于：（1）建立页表映射；（2）修改页表映射（如修改权限位）；（3）删除页表映射。

OSTD的页表操作依赖于`Cursor`或`CursorMut`，它们与指定的虚拟地址绑定。常用的页表操作包括：

- `map`：建立虚拟地址到物理地址的映射。需要传入 `PageProperty` 与 `UFrame`（代表物理内存页），映射建立后 Cursor 绑定的虚拟地址会自动前进 4K。

- `protect_next`：修改已建立映射的权限位。需要传入 `length`（修改范围）和 `operation`（权限修改操作）。
- `unmap`：取消已建立的映射。传入`length` 指定取消映射的范围。

OSTD的页表设计与实现较为复杂，感兴趣的同学可参考新发表的 SOSP 论文：[CortenMM: Efficient Memory Management with Strong Correctness Guarantees]

> **Rust的Closure（闭包）**
>
> 闭包是 Rust 中能够捕获上下文的匿名函数。与普通函数不同，闭包能自动推断参数和返回值类型，并通过三种 trait（`Fn`、`FnMut`、`FnOnce`）来明确对环境的捕获方式。例如：
>
> ```
> fn create_calculator(multiplier: i32) -> impl Fn(i32) -> i32 {
>  move |x| x * multiplier
> }
> fn main() {
>  let double = create_calculator(2);
>  let triple = create_calculator(3);
>
>  println!("Output: {}", double(5));  // Output: 10
>  println!("Output: {}", triple(5));  // Output: 15
> }
> ```

3. 用户空间

3.1 用户空间管理

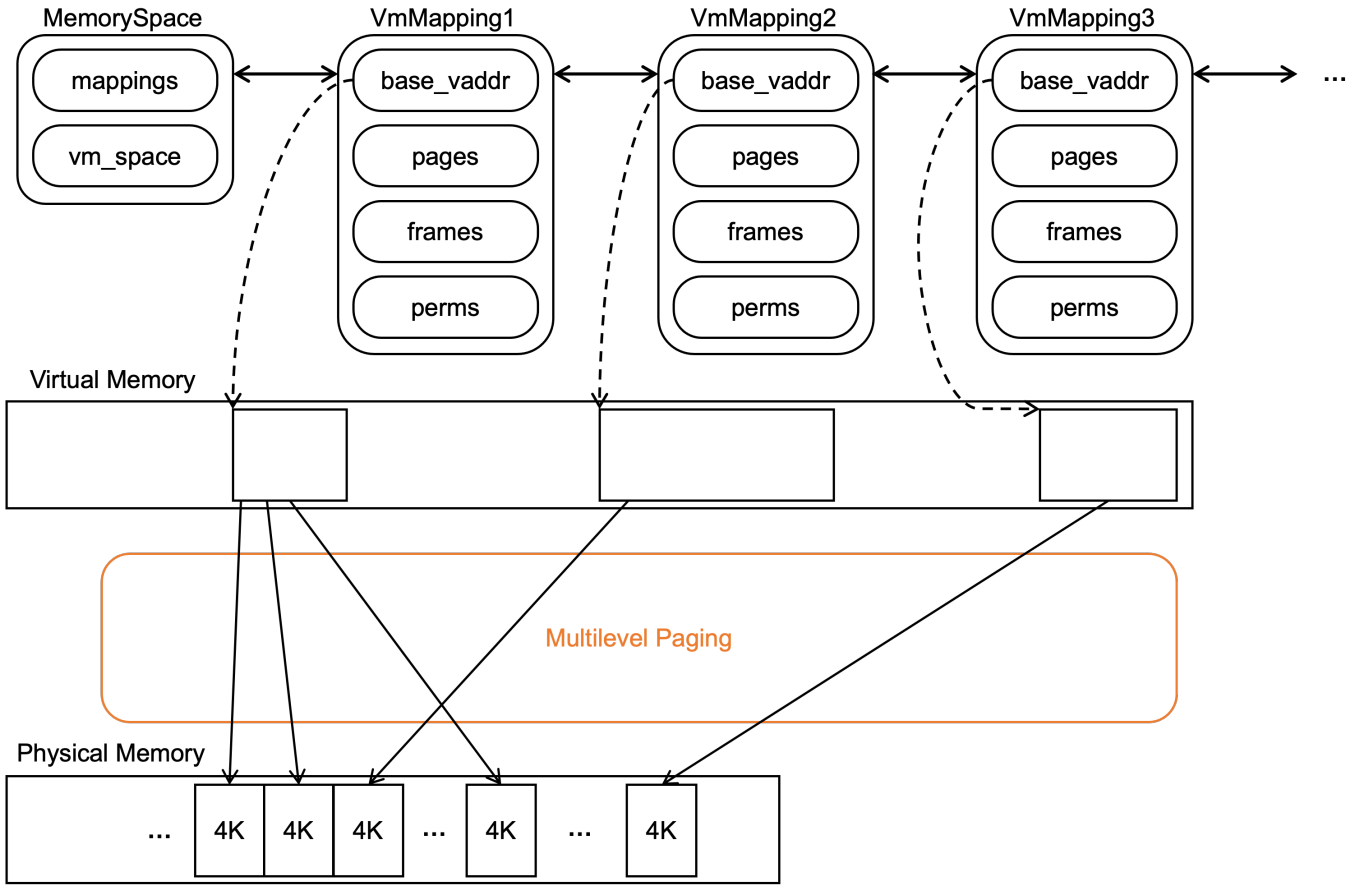
了解基本页表操作后，我们注意到 OSTD 的页表操作并未记录所有用户空间的映射信息。缺少这些信息，在执行 fork 等系统调用时需要遍历所有虚拟地址来查找映射，严重影响性能。因此需要设计 `MemorySpace` 结构来统一管理页表映射：

```
pub struct MemorySpace {
    vm_space: Arc<VmSpace>,
    mappings: SpinLock<LinkedList<VmMapping>>,
}

#[derive(Debug)]
pub struct VmMapping {
    base_vaddr: Vaddr,
    /// Mapping page count with PAGE_SIZE as unit.
    pages: usize,
    frames: Option<Segment<()>>,
    perms: PageFlags,
}
```

`MemorySpace` 除了存储 `VmSpace`，还通过 `mappings` 记录所有已建立的映射信息。`VmMapping` 存储特定映射的详细信息，包括起始虚拟地址、映射页数、权限以及底层物理页。

`MemorySpace`、`VmMapping`、虚拟内存和物理内存的对应关系如下：



`MemorySpace`主要提供两个函数：`map`以及`duplicate`：

1. `MemorySpace::map`：根据`VmMapping`信息自动分配物理页并建立映射，按虚拟地址从低到高的顺序返回物理页以便初始化数据。
2. `MemorySpace::duplicate`：用于 fork 系统调用，复制所有映射及底层物理页数据，生成完全一致的内存空间。

3.2 用户空间访问

在掌握用户空间管理的基本原理后，我们还需要了解如何安全地访问用户空间。这是因为许多系统调用会传入用户虚拟地址，这些地址指向协议指定的数据结构。例如，`read/write`系统调用中的地址指向字节数组，内核需要读取这些数据来完成相应的读写操作。

OSTD 为用户空间访问提供了一套友好的结构和接口——`VmReader`和`VmWriter`。这些结构依赖于与进程绑定的`VmSpace`，通过`reader`/`writer`函数可以获取指定地址对应的`VmReader`/`VmWriter`。以下代码展示了`sys_execve`中读取用户传入程序名的过程，通过获取`VmReader`将用户数据读取到内核缓冲区：

```
pub fn sys_execve(
    path: Vaddr, /* &[u8] */
    ...
) -> Result<SyscallReturn> {
    // The max file name: 255 bytes + 1(\0)
    let mut buffer = vec![0u8; 256];
    current_process
        .memory_space()
        .vm_space()
        .reader(path, MAX_FILENAME_LENGTH)
```

```
        .unwrap()
        .read_fallible(&mut VmWriter::from(&mut buffer as &mut [u8]))
        .unwrap();
    ...
}
```

这段代码展示了两个重要概念：

1. `read_fallible`：该方法会将数据复制到提供的`VmWriter`中。类似地，`VmWriter`也提供对应的`write_fallible`方法。
2. `VmWriter::from(&mut [u8])`：这是 OSTD 提供的便捷功能，允许将可变字节切片转换为`VmWriter`，其生命周期与原切片绑定。类似地，`VmReader`也提供`VmReader::from(&[u8])`转换方法。

VmReader / VmWriter 的错误处理机制

OSTD 为`VmReader`和`VmWriter`设计了两种状态标记：Fallible（可能失败）和 Infallible（不会失败）。

- **Fallible**：表示读写操作可能失败，主要用于用户空间访问。由于用户程序传入的虚拟地址可能没有有效映射或权限不足，访问过程中可能触发非法访问异常。
- **Infallible**：表示读写操作不会失败，主要用于内核空间访问。得益于 Rust 的内存安全特性，我们可以确保所有内核空间的访问都具有有效映射和适当权限，因此视为不可能失败。

这种设计使得内核能够安全地区分对待用户空间和内核空间的访问操作，保证对虚拟地址读写的安全性。

4. 上手练习

mprotect系统调用

在最开始支持标准库下RISC-V程序时，我们假装实现了mprotect系统调用（直接返回0），没有进行实际的操作。学习了页表操作后，我们就可以真正实现该系统调用，请你实现mprotect，根据用户传入的起始地址，长度，权限来修改映射。测试程序：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <stdint.h>

static volatile int UNUSED = 42;

int main()
{
    // mprotection to make the memory page containing UNUSED writable
    mprotect((void *)((uintptr_t)&UNUSED & ~(getpagesize() - 1)), getpagesize(), PROT_READ
| PROT_WRITE);

    // Modify the UNUSED variable
    *((int *)&UNUSED) = 100;

    // Print the modified value of UNUSED
    printf("Modified UNUSED value: %d\n", UNUSED);
```

```
    // Restore the memory page protection to read-only
    mprotect((void *)((uintptr_t)&UNUSED & ~(getpagesize() - 1)), getpagesize(),
PROT_READ);

    // Print the modified value of UNUSED again (Read OK)
    printf("Modified UNUSED value: %d\n", UNUSED);

    // Try writeing to UNUSED again (this should cause a segmentation fault)
    *((int *)&UNUSED) = 200;

    return 0;
}
```