0. 前言

实验概述

了解OSTD中内存页分配器的初始化与使用，及读写物理页的相关接口

实验内容

1. 了解OSTD中的内存页分配器初始化流程与使用
2. 了解OSTD中读写分配的空闲内存页相关接口

1. OSTD内存页分配器总览

1.1 初始化流程

OSTD中物理内存页分配器初始化涉及到两个步骤：

1. **内存分布信息获取。**OSTD根据启动协议和架构的不同，解析启动协议中传入的内存分布区域，并使用`non_overlapping_regions_from`来去除重叠区域。
2. **内存页分配器初始化。**OSTD根据内存分布区域，建立内核的物理内存分配器，并注册到全局的内存分配器中。

> **内存页分配器的调度算法**
>
> 空闲内存页的管理可以通过很多种算法，但OSTD目前仅限定了伙伴系统`buddy_system_allocator`进行可用物理内存页管理，且目前不允许我们像调度器一样进行注入操作，使得我们无法自定义一些实现算法如first fit和worst fit。

1.2 内存页分配与使用

OSTD提供了分配物理内存页的结构：`FrameAllocOption`。该结构的实例会存储一系列分配内存页所需要的基础信息如需要分配的大小，是否连续物理内存，是否需要清零操作。用户需要进行分配时则需要调用`alloc`或`alloc_xxx`接口来进行分配。依据不同的调用接口，分配的内存页会返回不同的结构实例，如`Frame`或`Segment`。实际上两者使用方式相同，均可以使用`read_xxx`或`write_xxx`来对分配的物理页进行读写操作。

2. 内存页分配器初始化与使用

本节将会更为细致的讲述初始化流程和内存页分配器的代码。

2.1 内存分布信息获取

在RISCV架构上，OSTD会使用OpenSBI作为bootloader进行加载，当bootloader进入到OSTD时，OSTD会根据bootloader传入的设备树物理地址进行解析，其内部会提供三种内存区域信息：（1）可用内存；（2）保留内存；（3）Initramfs内存。在系统启动后，会根据设备树提供的内存区域信息构建初始内存区域信息，并提供给OSTD全局使用：

```
fn init_memory_regions(memory_regions: &'static Once<Vec<MemoryRegion>>) {
    let mut regions = Vec::<MemoryRegion>::new();

```

```
    // Add the usable region.
    for region in DEVICE_TREE.get().unwrap().memory().regions() {
        ...
    }

    // Add the reserved region.
    if let Some(node) = DEVICE_TREE.get().unwrap().find_node("/reserved-memory") {
        ...
    }

    // Add the kernel region.
    regions.push(MemoryRegion::kernel());

    // Add the initramfs region.
    if let Some((start, end)) = parse_initramfs_range() {
        ...
    }

    memory_regions.call_once(|| non_overlapping_regions_from(regions.as_ref()));
}
```

在代码中除了使用到提供的三类信息外，还会排除内核代码和数据段所占据的内存，并在最后使用`non_overlapping_regions_from`剔除重叠区域，防止在分配可用内存访问权限时意外分配了内核数据段的访问权限。

2.2 内存页分配器初始化

在系统建立好初始内存区域信息后，会在某一个时间掉调用`ostd/src/mm/page/allocator.rs::init`来初始化内存页分配器

```
pub(crate) fn init() {
    let regions = crate::boot::memory_regions();
    let mut total: usize = 0;
    let mut allocator = FrameAllocator::<32>::new();

    for region in regions.iter() {
        if region.typ() == MemoryRegionType::Usable {
            // Make the memory region page-aligned, and skip if it is too small.
            let start = region.base().align_up(PAGE_SIZE) / PAGE_SIZE;
            let region_end = region.base().checked_add(region.len()).unwrap();
            let end = region_end.align_down(PAGE_SIZE) / PAGE_SIZE;
            if end <= start {
                continue;
            }

            // Add global free pages to the frame allocator.
            allocator.add_frame(start, end);
            total += (end - start) * PAGE_SIZE;
        }
    }
```

```
21
22      let counting_allocator = CountingFrameAllocator::new(allocator, total);
23      PAGE_ALLOCATOR.call_once(|| SpinLock::new(counting_allocator));
24  }
25
```

初始化期间会遍历所有内存区域，并将可用的内存区域进行对齐操作后，一并传入到`CountingFrameAllocator`中进行初始化操作，随后注册到全局的物理内存分配器中。

2.3 内存页分配

进一步查看`CountingFrameAllocator`，我们可以得到以下代码（去除了简单函数）：

```rust
/// FrameAllocator with a counter for allocated memory
pub(in crate::mm) struct CountingFrameAllocator {
    allocator: buddy_system_allocator::FrameAllocator,
    total: usize,
    allocated: usize,
}

impl CountingFrameAllocator {
    pub fn alloc(&mut self, count: usize) -> Option<usize> {
        match self.allocator.alloc(count) {
            Some(value) => {
                self.allocated += count * PAGE_SIZE;
                Some(value)
            }
            None => None,
        }
    }

    pub fn dealloc(&mut self, start_frame: usize, count: usize) {
        self.allocator.dealloc(start_frame, count);
        self.allocated -= count * PAGE_SIZE;
    }
}
```

该分配器会记录总的可用内存以及已经分配的内存量，其分配器算法使用的便是Rust开源库`buddy_system_allocator`提供的伙伴系统分配器。

> 伙伴系统 (buddy system) 资料来源：知乎
>
> 伙伴系统是一个结合了2的方幂个分配器和空闲缓冲区合并计技术的内存分配方案, 其基本思想很简单:
>
> 内存被分成含有很多页面的大块, 每一块都是2个页面大小的方幂。 如果找不到想要的块, 一个大块会被分成两部分, 这两部分彼此就成为伙伴. 其中一半被用来分配, 而另一半则空闲。 这些块在以后分配的过程中会继续被二分直至产生一个所需大小的块. 当一个块被最终释放时, 其伙伴将被检测出来, 如果伙伴也空闲则合并两者。

基于`CountingFrameAllocator`，OSTD进一步提供了`FrameAllocOption`来方便开发人员分配物理页访问对象，如`Frame`及`Segment`。

3. 内存页使用

以`Frame`作为例子，在获取内存页对象后，我们可以使用`read_xxx`或`write_xxx`来对分配的物理页进行读写操作，这是因为`Frame`基于所存储的物理地址，实现了`VmIo`的trait。`VmIo`只需要`Frame`提供读取和写入的基本接口，就可以自动实现`read_xxx`或`write_xxx`。

3.1 VmIo

`VmIo`是一个Rust的trait，其路径是`ostd/src/mm/io.rs`，重要的接口如下：

```
pub trait VmIo: Send + Sync {
    /// Reads requested data at a specified offset into a given `VmWriter`.
    fn read(&self, offset: usize, writer: &mut VmWriter) -> Result<()>;

    fn read_bytes(&self, offset: usize, buf: &mut [u8]) -> Result<()> {
        let mut writer = VmWriter::from(buf).to_fallible();
        self.read(offset, &mut writer)
    }

    fn read_val<T: Pod>(&self, offset: usize) -> Result<T> {
        let mut val = T::new_uninit();
        self.read_bytes(offset, val.as_bytes_mut())?;
        Ok(val)
    }

    /// Writes all data from a given `VmReader` at a specified offset.
    fn write(&self, offset: usize, reader: &mut VmReader) -> Result<()>;

    fn write_bytes(&self, offset: usize, buf: &[u8]) -> Result<()> {
        let mut reader = VmReader::from(buf).to_fallible();
        self.write(offset, &mut reader)
    }

    fn write_val<T: Pod>(&self, offset: usize, new_val: &T) -> Result<()> {
        self.write_bytes(offset, new_val.as_bytes())?;
        Ok(())
    }
}

```

在VmIo中，已经基于read和write接口默认实现了其它具有更高功能的接口，方便开发者进行使用，其中我们关注到接口有三个新内容：（1）`VmReader`；（2）`VmWriter`；（3）`Pod`。

> **POD数据类型**
>
> POD全称Plain Old Data，意为普通，旧的数据类型。该概念来源于C / C++，C++为了和旧的C数据进行兼容，提出了POD数据结构的概念，基本数据类型比如int，float，bool等是POD类型的。
>
> 在这里不会介绍具体POD类型的定义，只需要了解到OSTD将一系列基础数据类型如u8, u16...定义为了POD类型，并且只包含这些POD类型的结构或者枚举也可以通过`#[derive(Pod)]`来自动声明为POD类型，如此可以通过`read_val`，`write_val`和其它接口进行读写和转换。

3.2 VmReader & VmWriter

`VmReader`和`VmWriter`为OSTD定义的两个结构，其实例均会与一段连续的**虚拟地址**进行绑定，并分别提供了读接口和写接口。在了解它们的工作机制之前，需要知道OSTD对于虚拟地址空间的划分。

OSTD将虚拟地址空间划分为两大空间，分别为**高地址空间**（> `0xffff_8000_0000_0000`）和**低地址空间**（< `0x0000_8000_0000_0000 - PAGE_SIZE`）。高地址空间仅提供给内核使用，禁止用户态程序访问该空间；低地址空间允许用户态程序访问，我们写入的用户态程序代码，数据便是存放在该区域内。

了解OSTD的虚拟地址空间划分后，我们便可以通过源码查看工作逻辑，由于`VmReader`和`VmWriter`的逻辑类似，因此这里只会放`VmReader`的代码：

```

pub struct VmReader<'a, Fallibility = Fallible> {
    cursor: *const u8,
    end: *const u8,
    phantom: PhantomData<(&'a [u8], Fallibility)>,
}

impl<'a> VmReader<'a, Infallible> {
    pub unsafe fn from_kernel_space(ptr: *const u8, len: usize) -> Self {
        debug_assert!(len == 0 || KERNEL_BASE_VADDR <= ptr as usize);
        debug_assert!(len == 0 || ptr.add(len) as usize <=
KERNEL_END_VADDR);

        Self {
            cursor: ptr,
            end: ptr.add(len),
            phantom: PhantomData,
        }
    }

    pub fn read(&mut self, writer: &mut VmWriter<'_, Infallible>) -> usize {
        ...
    }

    pub fn read_val<T: Pod>(&mut self) -> Result<T> {
        ...
    }

    pub fn read_once<T: PodOnce>(&mut self) -> Result<T> {
        ...
    }

    pub fn to_fallible(self) -> VmReader<'a, Fallible> {
        unsafe { core::mem::transmute(self) }
    }
}

impl VmReader<'_, Fallible> {
    pub unsafe fn from_user_space(ptr: *const u8, len: usize) -> Self {
        debug_assert!((ptr as usize).checked_add(len).unwrap_or(usize::MAX)
<= MAX_USERSPACE_VADDR);

        Self {
            cursor: ptr,
            end: ptr.add(len),
            phantom: PhantomData,
        }
    }
```

```
47
48      pub fn read_val<T: Pod>(&mut self) -> Result<T> {
49          ...
50      }
51  }
```

可以看到`VmReader`实际上根据是否有读取失败的可能性，分成了两个不同的实现。对于`VmReader<'_, Infallible>`构造，其通过unsafe标记，要求传入的指针必须位于内核空间中；对于`VmReader<'_, Fallible>`，其通过unsafe标记，要求传入的指针必须位于用户空间中。除此之外，只存在`Infallible`到`Fallible`的转换。

如此进行标记的原因是，内核空间的虚拟地址读写在系统初始化期间已经建立好了虚拟地址到物理地址的映射，而用户空间的构建会依赖于用户态程序，有可能会碰到读写的过程中，缺少虚拟地址到物理地址映射的问题。

4. 上手练习

在之前的EXEC系统调用实现中，并没有实现`argv`和`envp`两个参数的解析以及支持，现在拥有了`VmReader`和`VmWriter`作为知识后，我们可以开始读取这两个参数值，为之后的解析做准备了（即使我们不了解虚拟地址的页表映射）。现请你在`sys_execve`系统调用中，解析`_argv_ptr_ptr`和`_envp_ptr_ptr`，使用`read_cstring`来读取这两个地址所存放的数据。

注意：

1. 请在使用`_argv_ptr_ptr`和`_envp_ptr_ptr`后，将前面的下划线去掉，加上下划线只是为了遵循未使用变量的命名规范。
2. 注意这两个值的名称后缀为`ptr_ptr`，是指针的指针，需要先获取指针地址后，再进行`read_cstring`操作，以NULL(0)作为结尾。
3. 需要特殊处理ptr为0的情况
4. 建议参考`sys_execve`中读取文件名的代码