# Lab12 - Kernel Space & Physical Memory Management

## 0. 前言

### 实验概述

本节实验将学习内核空间分布与物理内存分配的相关知识。我们将首先了解物理内存的初始化过程及相关代码，随后深入探讨其算法原理与安全保障机制。

### 实验代码

本节实验课没有新代码，可以继续使用lab11的代码。

## 1. 内核空间分布

OSTD，或Asterinas，将整个虚拟地址空间划分为两部分：低地址部分供用户态程序使用，高地址部分供内核使用，以此实现不同特权级之间的空间隔离。OSTD 在 `mm/kspace/mod.rs` 中定义了内核空间的分布布局，具体如下：

```
+-+ <- the highest used address (0xffff_ffff_ffff_0000)
| |         For the kernel code, 1 GiB.
+-+ <- 0xffff_ffff_8000_0000
| |
| |         Unused hole.
+-+ <- 0xffff_e100_0000_0000
| |         For frame metadata, 1 TiB.
+-+ <- 0xffff_e000_0000_0000
| |         For [`KVirtArea`], 32 TiB.
+-+ <- the middle of the higher half (0xffff_c000_0000_0000)
| |
| |
| |
| |         For linear mappings, 64 TiB.
| |         Mapped physical addresses are untracked.
| |
| |
| |
+-+ <- the base of high canonical address (0xffff_8000_0000_0000)
```

其会包含几个部分：

- 内核代码空间：存放内核代码。
- Frame元数据空间：存放物理页的元数据。
- `KVirtArea`空间：当内核需要建立新的页表映射（如 MMIO）时使用该空间。
- 线性映射区域：虚拟地址 = 物理地址 + offset（`0xffff_c000_0000_0000`），便于内核直接访问物理内存

本节课将介绍物理内存管理的接口、算法及其安全保障机制。

# 2. 物理内存管理

物理内存管理主要涵盖以下几个方面：物理内存的初始获取与初始化、物理内存页分配（采用伙伴系统），以及 Rust 堆内存管理（采用 Slab 分配器）。

## 2.1 物理内存分布信息获取

在 RISC-V 架构中，当 OpenSBI 跳转至 OSTD 启动内核时，OSTD 会解析 OpenSBI 传入的设备树，其中包含以下三种内存区域信息：（1）可用内存；（2）保留内存；（3）Initramfs 内存。系统启动后，将根据设备树提供的信息构建初始内存区域，供全局使用：

```
fn parse_memory_regions() -> MemoryRegionArray {
    let mut regions = MemoryRegionArray::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()).unwrap();
    // Add the initramfs region.
    if let Some((start, end)) = parse_initramfs_range() {
        ...
    }
    regions.into_non_overlapping()
}
```

获取内存分布信息后，OSTD 将其存入全局变量 `EARLY_INFO` 中。随后，`EarlyFrameAllocator` 将利用这些信息完成系统初步内存页分配器的初始化。最终，通过调用 `mm::frame::allocator::init()` 函数，系统将根据内存分布完成全局内存页分配器的初始化。

## 2.2 内存页分配与使用

开发人员可通过 `FrameAllocOptions` 调用 OSTD 的内存页分配功能（OSDK 提供了伙伴系统分配算法的实现 `osdk-frame-allocator`）。分配结果将返回 `Segment`（代表连续的物理内存页）或 `Frame`（代表单个物理内存页），两者均实现了 `VmIo` trait，以便上层进行物理内存的读写操作。

`FrameAllocOptions` 的关键代码如下：

```
impl FrameAllocOptions {
    pub fn alloc_frame_with<M: AnyFrameMeta>(&self, metadata: M) -> Result<Frame<M>> {
        let single_layout = Layout::from_size_align(PAGE_SIZE, PAGE_SIZE).unwrap();
        let frame = get_global_frame_allocator()
            .alloc(single_layout)
            .map(|paddr| Frame::from_unused(paddr, metadata).unwrap())
```

```
            .ok_or(Error::NoMemory)?;
        ...
        Ok(frame)
    }

    pub fn alloc_segment_with<M: AnyFrameMeta, F>(
        &self,
        nframes: usize,
        metadata_fn: F,
    ) -> Result<Segment<M>>
    where
        F: FnMut(Paddr) -> M,
    {
        let layout = Layout::from_size_align(nframes * PAGE_SIZE, PAGE_SIZE).unwrap();
        let segment = get_global_frame_allocator()
            .alloc(layout)
            .map(|start| {
                Segment::from_unused(start..start + nframes * PAGE_SIZE,
metadata_fn).unwrap()
            })
            .ok_or(Error::NoMemory)?;
        ...
        Ok(segment)
    }
}
```

`VmIo` 是定义在 `ostd/src/mm/io.rs` 中的 Rust trait，其主要接口如下：

```
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(())
    }
}
```

> **POD数据类型**
> POD（Plain Old Data）意为“普通旧数据”，该概念源自 C/C++。C++ 为兼容 C 数据格式而提出 POD 类型，基本数据类型如 `int`、`float`、`bool` 等均属于 POD 类型。
>
> OSTD 将一系列基础类型（如 `u8`、`u16` 等）定义为 POD 类型。仅包含这些 POD 类型的结构体或枚举也可通过 `#[derive(Pod)]` 自动声明为 POD 类型，从而可通过 `read_val`、`write_val` 等接口进行读写和类型转换。

## 2.3 内核堆管理

在 Rust 中，若在脱离 `std` 环境后仍需使用 `alloc` 相关库，则需实现 `GlobalAlloc` trait，并使用 `#[global_allocator]` 注册堆内存分配器。OSTD 在 `mm/heap/mod.rs` 中实现了该功能：

```
unsafe impl GlobalAlloc for AllocDispatch {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        ...
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        ...
    }
}
```

其中，`Layout` 包含内存区域的大小与对齐信息，`ptr` 表示起始地址。

由于代码中包含 `unsafe` 关键字，该段代码必须在 OSTD 中实现。Asterinas 在设计时将安全检查与分配逻辑分离：允许外部向 OSTD 注册符合指定 trait 的分配器实例，同时 OSTD 在实现 `GlobalAlloc` 时会调用已注册的堆分配器，并执行安全检查以确保操作正确。OSDK 默认提供了基于 Slab Allocator 算法的分配器实现（`osdk-heap-allocator`）。

# 3. 物理内存分配原理与安全保障

前文提到内核中的两种内存分配场景：一是以 4KB 为基本单位的内存页分配；二是针对小粒度、频繁分配释放的对象内存（如 8B、256B 等）。前者使用伙伴系统（Buddy System）进行管理，后者则使用 Slab 分配器，在伙伴系统提供的内存页基础上进一步优化小对象内存的分配。

为减少代码量，OSTD 并未完整实现伙伴系统和 Slab 分配器的所有逻辑，而是通过实现安全保障机制，确保内存分配不会引发安全问题，并将算法的主要逻辑置于 OSTD 外部。每种算法的安全保障机制将在相应章节简要介绍，更多细节请参阅 Asterinas 论文。

## 3.1 伙伴系统

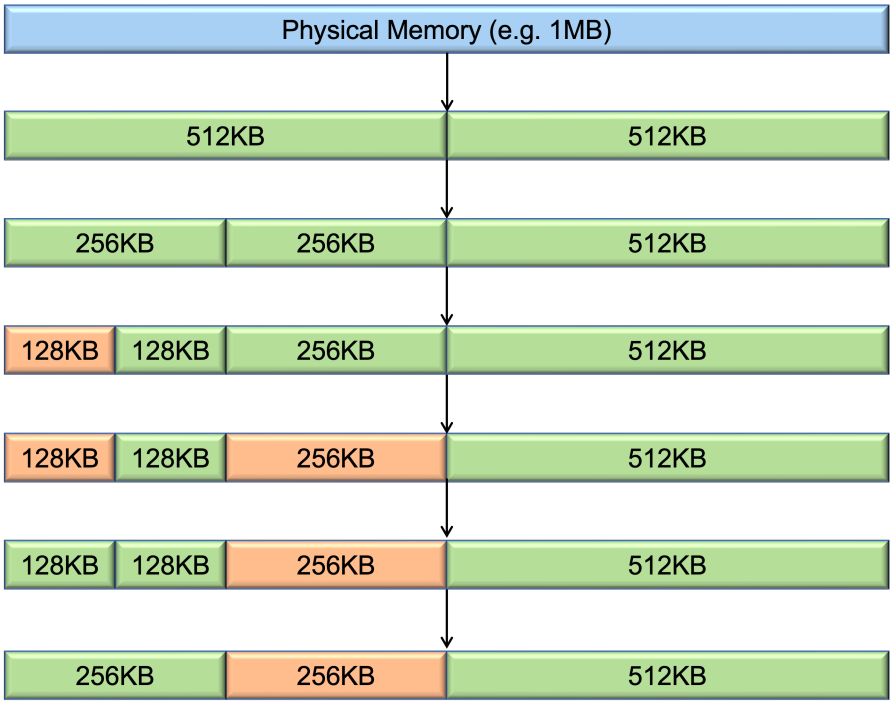
伙伴系统是一种经典的内存分配算法，它将整个物理内存视为一个二叉树，每个节点代表一个内存块。

**基本工作原理**：系统将可用内存划分为多个大小不同的块，每个块的大小为 2 的幂次方。当请求分配内存时，系统会寻找大小最合适的块。若未找到恰好匹配的块，则将较大的块对半分裂为两个“伙伴”块。该分裂过程递归进行，直至获得所需大小的块。

例如，假设存在 1MB 的连续内存空间。当应用程序请求 128KB 内存时，系统处理流程如下：

- 首先检查是否有 128KB 的可用块；
- 若没有，则将 1MB 块分裂为两个 512KB 的伙伴块；
- 再将其中一个 512KB 块分裂为两个 256KB 的伙伴块；
- 继续将其中一个 256KB 块分裂为两个 128KB 的伙伴块；
- 最终将其中一个 128KB 块分配给应用程序。

**伙伴系统的优势**在于其合并机制：当内存被释放时，系统会检查其伙伴块是否也为空闲状态。若是，则将这两个伙伴块合并为一个更大的块。下图展示了拆分与合并的示例：




然而，伙伴系统也存在局限性。由于要求块大小必须为 2 的幂次方，可能导致内部碎片。例如，申请 65KB 内存时，系统实际需分配 128KB 的块，造成 63KB 的空间浪费。

**安全保障**

安全保障分为分配与释放两个环节：

- **分配时**：确保分配的内存页处于空闲状态。内核为每个物理内存页维护引用计数，在 `Frame/Segment::from_unused` 中通过检查引用计数是否为 0，判断上层分配器分配的物理页是否空闲。
- **释放时**：确保释放后无人持有访问权限。利用 Rust 的生命周期与引用计数机制实现安全回收。

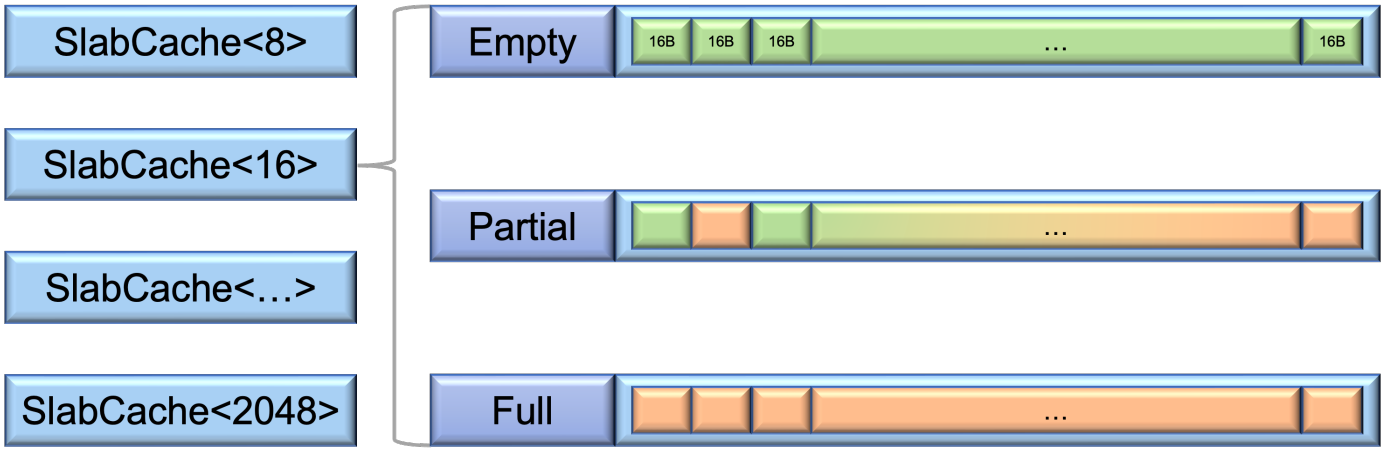
## 3.2 Slab Allocator

内核作为特殊应用程序，同样需要进行堆内存管理。伙伴系统最小分配单位为 4KB，对于内核中频繁分配释放的小对象（如 16B、128B 等）而言过于庞大，易导致资源浪费与管理复杂化。因此，需要设计适用于**小内存、高频分配释放**场景的新算法，即 Slab 分配器。

Slab 分配器的设计思想直观明了：预先准备小块内存以供快速分配。其核心组件包括：（1）缓存（Cache），用于分配特定大小的内存（如 8B、16B、128B 等）；（2）Slab，包含一个或多个物理内存页，存储多个相同大小的内存块。

Slab 分配器维护三种状态的 Slab：（1）满 Slab：所有内存块均已分配；（2）部分满 Slab：部分内存块已分配；（3）空 Slab：无任何内存块被分配。当系统需要分配内存块时，优先从部分满 Slab 中分配；若失败，则尝试空 Slab；若仍无法满足，则向伙伴系统申请新的物理内存页以创建新 Slab。

下图展示了 Slab 分配器的整体结构，包括 Cache、Slab 与内存块之间的关系：




**安全保障**

Slab 分配器的安全保障机制与伙伴系统有所不同，它进一步利用了 Rust 的强类型系统。与页分配器相比，OSTD 将堆内存分配器的分配与释放接口中的指针替换为 `HeapSlot`，代表上图中绿色或红色的内存区域。

图中的绿色或红色区域无法由上层开发人员直接创建，而需通过创建 `Slab` 并转换为 `SlabMeta` 后，通过 `SlabMeta::alloc` 函数从空闲列表中获取**已创建好的** **`HeapSlot`**。可以认为，上图中 Empty、Partial 与 Full 状态及其左侧部分属于 OSTD 外部，右侧则由 OSTD 内部管理。

该机制充分利用了 Rust 的强类型系统与所有权模型，具有以下优势：

- `HeapSlot` 的合法性由 OSTD 保障，确保其在创建时处于未被使用的状态。
- 当 `HeapSlot` 传入 OSTD 进行释放时，所有权模型确保释放后其他代码无法继续使用该 `HeapSlot`。

# 4. 上手练习

在之前的 RamFS 实现中，我们使用 `Vec<u8>` 存储数据，这会调用 Slab 分配器进行动态内存分配。虽然功能可行，但会带来较大的性能开销（每次扩展或收缩均需从 Cache 中分配）。请将 `Vec<u8>` 修改为 `Vec<Frame<()>>`，即使用一系列 4KB 内存页存储数据，并调整相应的读写逻辑。修改完成后，请重新运行 `ramfs` 程序以验证实现的正确性。