内存管理模块实现

内存布局

在 RISC-V 架构下，内核例程与用户例程均采用 Sv39 分页机制进行地址映射，内核与用户共享同一个页表结构（即用户映射与内核映射共存于同一页表，通过 PTEFlags 的 U 位进行隔离）。

- 内核空间 位于虚拟地址的高地址段，内核镜像起始处至可用内存的最高物理地址，按照固定偏移（KERNEL_OFFSET）映射至虚拟地址空间，便于内核直接通过已知偏移进行物理地址访问。
- 用户空间 位于虚拟地址的低地址段，用户程序镜像从低地址起映射，根据 ELF 文件加载。

该设计的一大优势在于：对于内核态代码而言，物理页帧号（PPN）与虚拟地址之间存在确定性偏移关系，内核访问物理页帧时无需额外查询页表，直接计算得到虚拟地址。这种机制极大地简化了页目录帧构建与写时复制（Copy-On-Write，COW）等内存管理操作。

在 Loongarch 架构下，内核例程与用户例程处于不同的映射模式下：

- 内核地址空间 通过 CSR_DMW0 窗口，采用直接映射（Direct Map Window，DMW），虚拟地址高段直接映射到物理地址，无需经过页表翻译，访问高性能。
- 用户地址空间 通过分页机制映射（通过设置 PWCL 和 PWCH 寄存器，使页表映射模式为 Sv39，与 riscv 一致，实现上层用户地址翻译的统一），虚拟地址低段需通过页表完成地址转换，支持动态映射、权限隔离等机制。

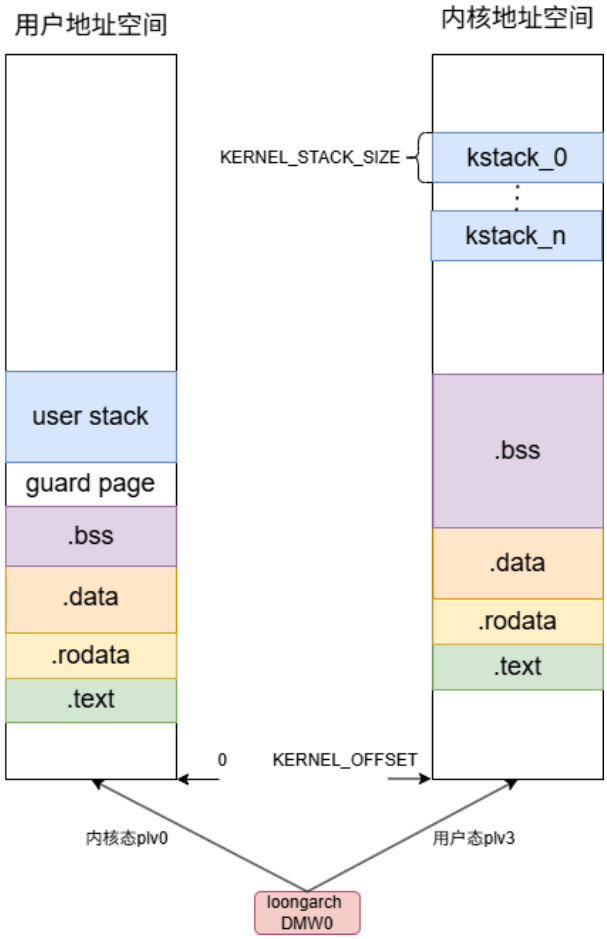

图 1: RocketOS 内存布局

堆分配器和用户页帧分配器

RocketOS 内核设计中，存在两个内存分配器，分别为 堆分配器 和 页帧分配器，其职责范围和实现机制有所不同：

- 堆分配器 负责内核动态内存的分配，管理位于内核镜像区域的内存空间，主要用于 kmalloc、对象创建等需要动态大小内存的场景。堆分配器对 buddy_system_allocator 进行封装， 基于 Buddy System 算法实现，支持不同大小块的合并与拆分，具备较好的内存利用率和碎片控制能力（如 代码 1 所示）。
- 页帧分配器 负责管理可用物理内存页帧，支持为页表映射、用户空间分配等场景按页分配物理内存。页帧分配器采用 stack-based bump allocator with recycling 策略，简洁高效，适合内核启动阶段以及高频页帧分配场景（如 代码 2 所示）。

两者的分配器设计均支持 RAII 资源管理，通过封装的跟踪器数据结构在对象生命周期结束时自动回收所占用的内存，简化了内存回收过程，提升了内核代码的健壮性和可靠性。

```
  pub struct Heap<const ORDER: usize> {
    free_list: [linked_list::LinkedList; ORDER],
    user: usize,
    allocated: usize,
    total: usize,
}
```

代码 1: 堆分配器结构

```
pub struct StackFrameAllocator {
    current: usize,
    end: usize,
    recycled: Vec<usize>,
}
```

代码 2: 页帧分配器结构

虚拟内存数据结构

一个虚拟地址空间由一个 `MemorySet` 对象描述（如代码 3 所示）。`MemorySet` 对象包含了唯一的地址空间标识、区域映射表和页表帧数组等。区域集合 `areas` 是 RocketOS 虚拟内存系统的核心数据结构，描述了虚拟地址空间的区域。不同的区域可能有不同的映射方式，例如线性映射、共享内存、写时复制等。页表帧数组用于存储页表帧的跟踪器。

```rust
pub struct MemorySet {
    /// 堆顶
    pub brk: usize,
    /// 堆底
    pub heap_bottom: usize,
    /// mmap 的起始地址，用于用户态 mmap
    pub mmap_start: usize,
    /// 页表
    pub page_table: PageTable,
    /// Elf, Stack, Heap, 匿名私有映射, 匿名共享映射, 文件私有/共享映射
    /// BTreeMap key 是 vpn_range 起始虚拟地址
    pub areas: BTreeMap<VirtPageNum, MapArea>,
    /// System V shared memory
    /// 映射: shm_start_address -> shmid
    pub addr2shmid: BTreeMap<usize, usize>,
}
```

代码 3: MemorySet 对象

虚拟地址空间区域实现

在 RocketOS 中，虚拟地址空间的区域管理由 MapArea 结构体承担(如[代码 4]所示)。它统一表示一段虚拟地址空间的区域，包括起始地址范围、权限、映射类型（如线性映射、按页分配、文件映射等），并提供了映射、取消映射、复制、扩展、分割等操作接口。

当前 RocketOS 支持的区域类型包括：

1. Linear：用于内核空间的线性映射；
2. Framed：按页分配物理页帧，适用于用户数据段等；
3. Stack：用户栈空间，通常按需分配；
4. Heap：用户堆空间，支持动态扩展；
5. Filebe：用户文件映射区域。

每个 MapArea 还包含一个页权限字段（MapPermission），支持读/写/执行/用户等权限标志，并可标记共享（S）或写时复制（COW）等属性。此外，MapArea 还支持关联文件映射及偏移量，便于实现 mmap 功能。

与整体地址空间解耦是 RocketOS 的一项重要设计。MapArea 并不持有页表根或页表帧等资源，而是通过传入的 PageTable 参数在映射和取消映射时完成操作。页帧的具体分配由上层的 MemorySet 管理。这种设计使得一个 MapArea 可被复制、移动并重用于不同的地址空间实例，有利于实现如 fork 中的地址空间复制、写时复制、懒分配高级功能。

```rust
pub struct MapArea {
    pub vpn_range: VPNRange,
    pub map_perm: MapPermission,
    pub pages: BTreeMap<VirtPageNum, Arc<Page>>,
    pub map_type: MapType,
    /// 文件映射
    pub backend_file: Option<Arc<dyn FileOp>>,
    pub offset: usize,
    pub locked: bool, // 是否被锁定，用于文件映射
}
```

代码 4: MapArea

线性映射区域

主要用于 内核地址空间 的直接映射，比如将物理内存映射到虚拟地址（通常是 KERNEL_BASE 起始）。

特点：

1. 虚拟页号（VPN）和物理页号（PPN）之间保持一个固定的偏移。（riscv 线性偏移，loongarch 直接映射）
2. 没有 Page 结构和物理页帧 Frame 的动态分配。
3. 权限固定、不可更改。
4. 映射时通过 `map_range_continuous` 一次性完成, 占用连续的物理空间。

独占物理页帧映射

用于用户进程的私有映射，如代码段、数据段、匿名映射等。

特点：

1. 每个页都由 `Page::new_framed` 分配一个新页帧。
2. 拥有独立物理页帧的写权限。
3. 可在 fork 时转变为 CopyOnWrite（写时复制）逻辑。

栈区域

表示用户进程的栈内存区域，支持懒分配和向下增长。

特点：

1. 栈空间按页分配，初始时不分配物理页帧 o
2. 当访问未映射页时触发 page fault，缺页异常处理器会分配一页新帧，并更新页表。
3. 当访问栈空间下的保护页时,， 会将整个区域的 vpn_range.start 向下扩展一页，从而“向下增长”栈空间。
4. 为防止栈无限扩张，会检查其与前一个区域之间的页数是否大于设定的 栈保护间隙（STACK_GUARD_GAP_PAGES），若间隙不足则终止扩张并触发 SIGSEGV。

堆区域

用户堆映射，支持扩展, 堆顶为 `brk`，堆底为 `heap_bottom`。与 `Framed` 共享很多行为，但更强调向上的扩展性

文件映射区域

用户通过 mmap 创建的文件映射区域。

特点：

1. 访问时延迟加载页帧（除非 MAP_POPULATE 会预分配）由 `backend_file.get_pages()` 获取物理页帧。
2. 页帧内容可能会被写回文件。

共享内存区域

RocketOS 支持两种类型的共享内存：System V 共享内存和匿名共享内存。System V 共享内存由 `shmat` 系列系统调用产生，匿名共享内存由 `mmap` 系统调用产生。System V 共享内存通过 `ShmSegment` 对象管理，`SysVShm` 对象包含了共享内存的 id 和对应分配的页帧，如代码 5 所示。使用 Weak 而非 Arc 是为了将物理页的生命周期交由进程 attach/detach 控制，体现了“共享但非所有”的语义。

```rust
pub struct ShmSegment {
    pub id: ShmId,                  // 段的元数据
    pub pages: Vec<Weak<Page>>,  // 共享页帧列表（使用 Weak 管理引用）
}
```

代码 5: ShmSegment

缺页处理

RocketOS 支持两种可恢复的缺页异常处理机制：写时复制（Copy-on-Write, COW） 和 懒分配（Lazy Allocation）。

写时复制

写时复制机制用于进程创建后共享只读内存页（例如通过 fork），其页表项包含 COW 标志。当子进程试图写该页时： 若该页只被一个进程引用，直接清除 COW 标志，赋予写权限； 否则，分配一个新物理页，将原页内容复制到新页，并更新页表项。

算法 0-0: 写时复制页故障处理逻辑

input: va, page_table, area
output: Updated page table and area.pages

```
pte ← page_table.find_pte(va)
if pte.flags has COW then
    page ← area.pages[va]
    if refcount(page) == 1 then
        pte.flags.remove(COW)
        pte.flags.insert(W)
        page_table.update(va, pte)
    else
        new_page ← alloc_frame()
        memcpy(new_page, page)
        pte.flags.remove(COW)
        pte.flags.insert(W)
        page_table.update(va, new_page)
        area.pages[va] ← new_page
    end
else return SIGSEGV
```

懒分配

懒分配机制适用于匿名区域（如堆、栈）和文件映射区域（如 mmap），延迟到首次访问再分配实际物理页。

- 对于文件映射区域（Filebe）：
  - 若为共享或只读映射，直接通过 backend 提供物理页；
  - 若为私有写映射，执行写时复制操作；
- 对于匿名区域，如 Heap：
  - 缺页时分配页帧，并更新页表；
  - 为优化效率，会批量分配最多 4 页；
- 对于 Stack 区域：
  - 当访问的是栈底 VPN，会尝试向下增长一页，并更新 vpn_range。

算法 **0-0:** 懒分配页故障处理逻辑

```
    input: va, area, cause
    output: Updated page table and area.pages
1   vpn ← floor(va)
2   if area.type = Filebe then
3       offset ← area.offset + (vpn - area.start) * PAGE_SIZE
4       page ← area.backend.get_page(offset)
5       if page exists then
6           if cause = STORE and !area.is_shared then
7               new_page ← copy_frame(page)
8               map(va, new_page, Writable)
9               area.pages[vpn] ← new_page
10          else
11              map(va, page, area.perm)
12              area.pages[vpn] ← page
13          end
14      else return SIGBUS
15  else if area.type = Stack then
16      if vpn = area.start then
17          if prev_area exists and !guard_gap_ok then return SIGSEGV
18          area.start ← area.start - 1
19          self.areas.update_key(area)
20      new_page ← alloc_frame()
21      map(va, new_page, Writable)
22      area.pages[vpn] ← new_page
23  else
24      for i in [vpn .. vpn+4] do
25          if !area.pages.contains(i) then
26              page ← alloc_frame()
27              map(i, page, Writable)
28              area.pages[i] ← page
29          end
30  end
```

通过上述设计，RocketOS 能够在创建进程时实现写时复制，减少复制开销；同时允许了程序申请巨大的内存空间，而不会立即分配物理页帧，提高了内存分配的效率。