2025春夏季开源操作系统训练营总结报告

王艳东

大页分配

- 大页内存（Huge Pages）是指操作系统为提高内存管理性能而使用的一种内存分配技术。通过将多个小页组合成一个大页（如2MB、1GB等），可以减少页表项的数量，降低CPU访问内存时的页表查找开销，从而提高内存访问效率。
- 大页内存的类型：
  - 标准大页：需要手动配置，适用于GB级别的内存分配
  - 透明大页：由内核自动管理，适用于TB级别的内存，减少了手动管理的复杂性

内存分配流程

- 应用程序请求内存
- 查找可用内存：AddrSpace::find_free_area()
- 分配和映射内存区域：AddrSpace::map_alloc()
  - Populate=true => 通过Backend::map_alloc()方法立即分配物理页面
  - Populate=false => 延迟分配，创建虚拟内存区域并等待页错误，触发页错误之后通过Backend:: handle_page_fault_alloc()方法分配内存
- 分配物理页帧：Backend:: alloc_frame()方法，通过调用全局分配器分配一段连续的物理内存

查找内存

```
let mut last_end: VirtAddr = hint.max(limit.start).align_up(align);
for area: &MemoryArea<Backend> in self.areas.iter() {
    if area.end() <= last_end {
        last_end = last_end.max(area.end().align_up(align));
    } else {
        break;
    }
}
```

初始化搜索位置

查找空闲间隙

```
for area: &MemoryArea<Backend> in self.areas.iter() {
    let area_start: VirtAddr = area.start();
    if area_start < last_end {
        continue;
    }
    if last_end VirtAddr
        .checked_add(size) Option<VirtAddr>
        .is_some_and(|end: VirtAddr| end <= area_start)
    {
        return Some(last_end);
    }
    last_end = area.end().align_up(align);
}
```

```
if last_end VirtAddr
    .checked_add(size) Option<VirtAddr>
    .is_some_and(|end: VirtAddr| end <= limit.end)
{
    Some(last_end)
} else {
    None
}
```

检查末尾空间

分配物理页帧

```rust
pub(crate) fn alloc_frame(zeroed: bool, align: PageSize) -> Option<PhysAddr> {
    let page_size: usize = align.into();
    let num_pages: usize = page_size / PAGE_SIZE_4K;
    let vaddr: VirtAddr = VirtAddr::from(global_allocator().alloc_pages(num_pages, align_pow2: page_size).ok()?);
    if zeroed {
        unsafe { core::ptr::write_bytes(dst: vaddr.as_mut_ptr(), val: 0, count: page_size) };
    }
    let paddr: PhysAddr = virt_to_phys(vaddr);

    #[cfg(feature = "cow")]
    frame_table().inc_ref(paddr);

    Some(paddr)
}
```

以4KB为单位，根据对齐参数大小，计算页面数量`num_pages`，调用全局分配器，分配num_pages个连续的物理页面，将这段连续内存的地址返回并映射到页表中，作为一整个页面。

取消映射

```rust
pub fn unmap(&mut self, start: VirtAddr, size: usize) -> AxResult {
    self.validate_region(start, size, align: PageSize::Size4K)?;

    let end: VirtAddr = start + size;
    for area: &MemoryArea<Backend> in self &mut AddrSpace
        .areas MemorySet<Backend>
        .iter() impl Iterator<Item = &MemoryArea<…>>
        .skip_while(move |a: &&MemoryArea<Backend>| a.end() <= start) impl Iterator<Item =…
        .take_while(move |a: &&MemoryArea<Backend>| a.start() < end)
    {
        let area_align: PageSize = match *area.backend() {
            Backend::Alloc { populate: _, align: PageSize } => align,
            Backend::Linear {
                pa_va_offset: _,
                align: PageSize,
            } => align,
        };

        let unmap_start: VirtAddr = start.max(area.start());
        let unmap_size: usize = end.min(area.end()) - unmap_start;
        if !unmap_start.is_aligned(area_align) || !is_aligned(addr: unmap_size, area_align.into()) {
            return ax_err!(InvalidInput, "address not aligned");
        }
    }

    self.areas MemorySet<Backend>
        .unmap(start, size, page_table: &mut self.pt) Result<(), MappingError>
        .map_err(op: mapping_err_to_ax_err)?;
    Ok(())
} fn unmap
```

首先调用 validate_region 验证要取消映射的地址范围是否有效，使用 4K 页面大小作为基本对齐要求。计算结束地址 end = start + size，然后遍历所有与要取消映射范围重叠的内存区域 。对于每个重叠的内存区域，函数提取其后端的对齐要求，然后计算实际需要取消映射的区域。如果 unmap_start 不满足区域对齐要求，或者 unmap_size 不是对齐大小的倍数，函数返回 InvalidInput 错误。通过验证后，调用 self.areas.unmap() 方法实际执行取消映射操作，传入起始地址、大小和页表的可变引用

基于buddy算法的页分配器

- 初始化过程：将起始地址和结束地址对齐到页边界，内存基址对齐到1G边界（与arceos的BitmapPageAllocator保持一致），将整个内存区域分解为最大可能的2的幂次个块。
- 分配算法：检查需要分配的页面数量和页面大小，验证其是否与PAGE_SIZE对齐，根据所需页面数量和对齐，计算所需阶数，查找可用块，将大块分割到所需大小，将多余部分加入对应的空闲链表，标记页面为已分配，更新使用统计。
- 释放与合并算法：首先标记当前块为空闲，然后递归检查伙伴块是否空闲，如果是则合并，将最终合并的块加入到对应阶数的空闲链表。

BitmapPageAllocato vs BuddyPageAllocator

```
Bitmap coalescing efficiency: 100.0%
Buddy coalescing efficiency: 100.0%
test tests::benchmark_tests::test_coalescing_efficiency ... ok
Random Small Pattern:
  Bitmap: 50.0%, Buddy: 83.3%
Mixed Size Pattern:
  Bitmap: 41.7%, Buddy: 33.3%
Power-of-2 Pattern:
  Bitmap: 16.7%, Buddy: 33.3%
test tests::benchmark_tests::test_fragmentation_comparison ... ok
```

合并效率测试结果：两个分配器都显示了 100% 的合并效率，这意味着它们都能完美地将释放的单页重新合并成大的连续内存块

碎片化测试结果分析：
- Random Small Pattern（随机小块分配）：
  - Bitmap: 50.0% 碎片化率
  - Buddy: 83.3% 碎片化率
  - BitmapPageAllocator 表现更好
- Mixed Size Pattern（混合大小分配）：
  - Bitmap: 41.7% 碎片化率
  - Buddy: 33.3% 碎片化率
  - BuddyPageAllocator 表现更好
- Power-of-2 Pattern（2的幂次分配）：
  - Bitmap: 16.7% 碎片化率
  - Buddy: 33.3% 碎片化率
  - BitmapPageAllocator 表现更好

BitmapPageAllocator在随机小块分配和2的幂次分配模式下碎片化率显著更低，BuddyPageAllocator 只在混合大小分配模式下表现更好，这符合buddy算法的设计特点——更适合处理不同大小的内存块分。，ArceOS选择 BitmapPageAllocator 作为默认页分配器是合理的，因为它在大多数实际使用场景下都能提供更好的内存利用率和更低的碎片化

感谢观看