SLUB设计文档

SLUB 核心思想

采用两级内存管理架构，底层使用页分配器进行整页分配，上层在页内**构建对象缓存**，将每个页划分为多个固定大小的对象并通过位图跟踪分配状态，从而实现小内存对象的快速分配和释放，同时通过整页回收机制避免内存碎片。

缓存（Caches）

SLUB 为**每种大小的内存对象维护一个缓存（cache）**，每个缓存对应一种特定大小的对象，管理着这些对象的分配和释放。其中**每个缓存管理一些slab**，形成slab链表，并且cache中的对象大小固定，可以减少内存碎片并提高分配效率。

Slab管理

一个 slab 是**一块连续的内存区域**，本实验中我们以页为单位来构造slab，其中包含多个相同大小的对象，并且每个 slab 都与一个特定的 cache相关联。

在状态管理方面，每个 slab 可以处于三种状态之一：

- **完全空闲**：所有对象都未被分配。
- **部分分配**：部分对象已被分配，部分对象仍然空闲。
- **完全分配**：所有对象都已被分配。

对象的分配和释放

分配对象：

- 当需要分配一个对象时，SLUB 会在对应缓存的 slabs 中寻找第一个有空闲对象的 slab。
- 如果找到一个部分分配的 slab，SLUB 会分配一个空闲对象并更新 slab 的状态。
- 如果没有找到合适的 slab，SLUB 会创建一个新的 slab 并分配对象。

释放对象：

- 当释放一个对象时，SLUB 会将其返回到所属的 slab，并更新 slab 的状态。
- 如果一个 slab 中所有对象都被释放，SLUB 可以将该 slab 返回给内存池以供重用。

设计实现

1.设计思路

我们实现的 SLUB 算法采用**两层架构**的高效内存单元分配：

- **第一层**：基于页大小的内存分配，使用 Best-Fit 算法作为底层页分配策略
- **第二层**：在第一层基础上实现基于任意大小的内存分配，专门处理小内存请求

核心设计：

- 小内存需求（≤128B）由 SLUB 管理，大内存需求（>128B）回退到页分配器
- 每个 slab 占用一个物理页（4KB），内存布局精心设计

- 使用位图高效跟踪对象分配状态

2. 内存布局设计

每个 slab 页面的内存布局如下：

```
[slab头][对象0][对象1]...[对象N][位图]
```

详细结构：

- **slab头**： `slab_t` 结构体，管理 slab 元数据
- **对象数组**：连续存储的固定大小对象
- **位图**：每个位对应一个对象的分配状态（0=空闲，1=已分配）

3. 核心数据结构

Slab 结构

```
typedef struct slab {
    list_entry_t list_link;     // 链接到缓存中的 slab 列表
    void *objects;              // 对象数组起始地址
    unsigned char *bitmap;      // 对象分配位图
    size_t free_objects;        // 空闲对象数量
    size_t total_objects;       // 总对象数量
```

```
    struct cache *cache;        // 所属缓存
} slab_t;
```

Cache 结构

```
typedef struct cache {
    list_entry_t slabs_full;    // 全满的 slab 列表
    list_entry_t slabs_partial; // 部分分配的 slab 列表
    list_entry_t slabs_free;    // 完全空闲的 slab 列表
    size_t object_size;         // 对象大小
    size_t objects_per_slab;    // 每个 slab 的对象数量
    const char *name;           // 缓存名称
} cache_t;
```

4. 初始化机制

两层初始化架构:

第一层：页分配器初始化

- 使用 Best-Fit 算法初始化物理页管理
- 建立完整的物理内存管理框架

第二层：SLUB 缓存初始化

```
static cache_t slub_caches[SLAB_CACHE_NUM];  // 3个固定大小缓存

void slub_init(void) {
    size_t sizes[SLAB_CACHE_NUM] = {32, 64, 128};
    const char *names[SLAB_CACHE_NUM] = {"size-32", "size-64", "size-128"};

    for (int i = 0; i < SLAB_CACHE_NUM; i++) {
        // 初始化每个缓存
        cache_init(&slub_caches[i], sizes[i], names[i]);
    }
}
```

对象数量计算算法:

通过求解不等式计算每个 slab 能容纳的最大对象数：

```
slab_struct_size + x * object_size + ceil(x/8) ≤ 4096
```

计算结果：

- 32B 对象：125个/slab
- 64B 对象：63个/slab
- 128B 对象：31个/slab

5. 分配算法

分配流程

```
void *slub_alloc(size_t size) {
    // 1. 延迟初始化检查
    if (!slub_initialized) slub_init();

    // 2. 大对象回退检查
    if (size > 128) return slub_alloc_pages(...);

    // 3. 查找合适缓存
    cache_t *target_cache = find_suitable_cache(size);

    // 4. 三级查找策略
    slab_t *slab = NULL;
    if (!list_empty(&target_cache->slabs_partial)) {
        // 优先使用部分分配的slab
        slab = get_from_partial_list(target_cache);
    } else if (!list_empty(&target_cache->slabs_free)) {
        // 其次使用空闲slab
        slab = get_from_free_list(target_cache);
    } else {
        // 最后创建新slab
        slab = slab_create(target_cache);
    }

    // 5. 在位图中查找空闲对象
    for (每个对象位置) {
        if (位图显示空闲) {
```

```
                设置位图标记；
                更新空闲计数；
                返回对象地址；
            }
        }
    }
```

三级查找策略

1. **部分分配slab优先**：最高效，直接分配空闲对象
2. **空闲slab次之**：需要移动到部分分配列表
3. **创建新slab**：最后手段，分配新物理页

6. 释放算法

释放流程

```
void slub_free(void *obj) {
    // 1. 找到对象所属的slab
    struct Page *page = pa2page(PADDR(obj));
    slab_t *slab = (slab_t *)page2kva(page);

    // 2. 计算对象索引
    offset = obj - slab->objects;
    index = offset / object_size;
```

```
    // 3. 更新位图和计数
    slab->bitmap[index] = 0;  // 标记为空闲
    slab->free_objects++;

    // 4. 状态转移处理
    list_del(&slab->list_link);
    if (slab->free_objects == slab->total_objects) {
        // 完全空闲 → 移动到空闲列表
        list_add(&cache->slabs_free, &slab->list_link);
    } else {
        // 保持部分分配状态
        list_add(&cache->slabs_partial, &slab->list_link);
    }
}
```

7. 状态管理机制

Slab 状态转移图

```
完全空闲 (slabs_free)
    ↑↓ 分配/释放对象
部分分配 (slabs_partial)
    ↑↓ 分配所有对象/释放到全空
完全占用 (slabs_full)
```

状态转换规则

分配时：

- 空闲 → 部分分配（首次分配）
- 部分分配 → 完全占用（分配最后一个对象）

释放时：

- 完全占用 → 部分分配（释放一个对象）
- 部分分配 → 完全空闲（释放到全空）

测试部分

我们针对SLUB算法的功能进行了**精简测试**，测试主要覆盖以下四个方面：

- 基本分配释放功能
- 内存重用机制
- 大对象回退机制
- 多对象管理和数据完整性

具体测试代码与测试结果见slum_pmm.c最后部分以及终端运行结果。