Challenge1设计文档

概述

Buddy 内存管理器用于管理物理内存，将物理页按照 2 的幂次划分为不同大小的块，实现高效的分配、释放和合并。每个空闲块维护在对应阶数的双向链表中，分配时可以快速找到合适大小的空闲块，释放时可以自动合并相邻伙伴块。

核心设计目标

1. 空闲块按阶数组织；
2. 支持快速分配、释放与合并；
3. 检查保证功能的正确性；

主要使用的数据结构

- 首先肯定是在memlayout.h中定义的`Pages`

```
struct Page {
    int ref;                        // page frame's reference counter 引用计数
    uint64_t flags;                 // array of flags that describe the status of
the page frame 页状态标志
    unsigned int property;          // the num of free block, used in first fit
pm manager 仅头页使用，表示页大小
    list_entry_t page_link;         // free list link 空闲链表链接
};
```

- `free_lists`

```
static list_entry_t free_lists[BUDDY_MAX_ORDER];
```

> 每个阶数维护一个双向循环链表，存储大小为 2^{order} 页的空闲块。
>
> `max_order_inited` 记录初始化过程中实际出现过的最大阶数。

- 一些宏定义和工具函数

`block_pages_from_order(o)`：方便计算2^o；

`page_index(p)`：计算页号；

`buddy_of(p, order)`：通过 XOR 计算伙伴页指针。

> **找伙伴的核心思想：**
>
> - 每个空闲块的大小是 2^{order}页。
> - 每个空闲块都有一个 **伙伴块**（buddy），大小相同，并且它们可以 **合并成更大的块**。
>
> 合并规则：
>
> - 两个块连续且大小相同时，它们可以合并。
> - 分配或释放时，需要快速找到 buddy 块。
>
> ***那么如何判断哪一个块是可以合并的buddy块呢?***

> 在 buddy 系统中：
>
> - 块大小 = 2^{order}页
> - 块的起始页索引 `idx` 必须满足 **对齐要求**：
>
> $$idx \ mod \ 2^{order} = 0$$
>
> - 这保证了每个块的起始索引是 **2^{order} 的倍数。**
>
> 那么，当我们的索引用二进制表示的时候，它的低order位一定都是0。
>
> 可以合并的两个buddy块的索引满足，
>
> $$左块idx = k \times 2^{order}$$
> $$右块idx = 左块idx + 2^{order}$$
>
> 也就是说两块的`idx`的二进制表示只会在第order位不同，那么如果要找到`buddy`的idx只需要反转第order位就可以了。
>
> 所以只需要对索引的第 `order` 位进行 **XOR 操作**：
>
> $$buddy\_idx = idx \oplus 2^{order}$$

函数设计

1. `buddy_init_memmap`

```
static void buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);

    size_t base_idx = page_index(base);
    size_t remain = n; //剩余没有划分的页数

    while (remain > 0) {
        /* 对当前位置能支持的最大对齐块长度进行选择 */
        unsigned int order = 0;
        /* 找到最大 order 满足：(1) block size <= remain；(2) base_idx % block_size
== 0 */
        for (int o = 0; o < BUDDY_MAX_ORDER; ++o) {
            size_t blocksz = (1UL << o);
            if (blocksz > remain) break;
            if ((base_idx % blocksz) == 0) order = o;
        }
        /* 将该块加入 free list */
        struct Page *head = &pages[base_idx];
        head->flags = head->property = 0;
        set_page_ref(head, 0);
        add_block_to_freelist(head, order);

        base_idx += (1UL << order);
        remain -= (1UL << order);
    }

    /* 更新 max_order_inited */
```

```
    for (int o = BUDDY_MAX_ORDER - 1; o >= 0; --o) {
        if (!list_empty(&free_lists[o])) {
            max_order_inited = (unsigned int)o;
            break;
        }
    }
}
```

目标：把传入的连续区域 `[base, base+n)` 切分成**尽可能大的且对齐的** 2^k 大小块并加入相应 free_list。

核心步骤：

- 对当前 `base_idx`，寻找最大的 `order` 使得 `blocksz<=remain` 且 `base_idx % blocksz == 0`（即起始地址满足该大小的对齐）。
- 将该对齐块作为一个空闲块加入 `free_lists[order]`，然后跳过这部分（`base_idx += blocksz`）。

这样能保证每个加入的块都**满足 buddy 对齐规则**（便于后续用 XOR 找伙伴）。

2. `buddy_alloc_pages`

```
static struct Page *buddy_alloc_pages(size_t n) {
    assert(n > 0);

    /* 计算所需 order（最小使 2^order >= n） */
    unsigned int need_order = 0;
    while ((1UL << need_order) < n) ++need_order;
    if (need_order >= BUDDY_MAX_ORDER) return NULL;//无足够内存

    /* 找到第一个非空的 free_list，从 need_order 到 max */
    unsigned int o;
    for (o = need_order; o < BUDDY_MAX_ORDER; ++o) {
        if (!list_empty(&free_lists[o])) break;
    }
    if (o == BUDDY_MAX_ORDER) {
        return NULL; //无足够内存
    }

    /* 从 order o 拿一个块 */
    list_entry_t *le = list_next(&free_lists[o]);
    struct Page *blk = le2page(le, page_link);
    remove_block_from_freelist(blk, o);

    while (o > need_order) {
        --o;
        /* 拆分：blk 大小为 2^(o+1)，拆成 blk (低半) 和 buddy (高半) 两个 2^o */
        struct Page *right = blk + (1UL << o); /* 右半作为空闲插入 */
        /* 初始化右半块头并插入 o 链表 */
        right->flags = right->property = 0;
        set_page_ref(right, 0);
        add_block_to_freelist(right, o);
        /* blk 保持为低半并继续拆分（不需要更新 blk） */
    }
```

```
    /* blk 为最终分配块的头 */
    /* 标记为已分配：把 PG_reserved 置位，清除 PG_property */
    ClearPageProperty(blk);
    blk->property = 0;
    for (size_t i = 0; i < (1UL << need_order); ++i) {
        SetPageReserved(blk + i);
        set_page_ref(blk + i, 0);
    }

    return blk;
}
```

流程：

1. 计算最小满足 `n` 页的 `need_order`，即 2^{order}`>= n`。
2. 在 `free_lists` 从 `need_order` 向上找第一个非空阶 `o`。
3. 从 `free_lists[o]` 取出一个块 `blk`（头），并 `remove` 它。
4. 若 `o > need_order`，则**向下拆分**：每次拆分把右半块（高地址半块）作为新空闲块放入 `free_lists[o-1]`，并把 `o` 减 1，直到 `o == need_order`。
    - 拆分方式用了 `right = blk + (1<<o-1)`（代码中写为 `1<<o` 之后先 `--o` 等价）。拆分后 `blk` 保持作为低半头用于继续拆分或分配。
5. 最终 `blk` 为分配给调用者的块头：清除 `PG_property`、设置 `PG_reserved`（把分配的每页标记为已占用），并返回 `blk`。

重要：分配返回的是**连续的 2^need_order 页**（块大小），即分配粒度是 `2^k`。如果用户请求的 `n` 不是精确的 2 的幂，这里仍然会返回 `2^k` 大小（满足不小于请求）。

3. `buddy_free_pages`

```
static void buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    /* 找到 order */
    unsigned int order = 0;
    while ((1UL << order) < n) ++order;
    assert((1UL << order) == n); /* 释放的块大小应为 2^order */

    /* 清除 reserved 标志（表示可用） */
    for (size_t i = 0; i < n; ++i) {
        ClearPageReserved(base + i);
        set_page_ref(base + i, 0);
        ClearPageProperty(base + i);
        (base + i)->property = 0;
    }

    struct Page *head = base;
    /* 尝试合并：循环直到达到最大 order 或无法合并 */
    while (order < BUDDY_MAX_ORDER - 1) {
        struct Page *b = buddy_of(head, order);
        /* 如果 buddy 是空闲头并且大小与当前阶相同，则可合并；如果不是就不可以 */
        if (!PageProperty(b) || b->property != (1UL << order)) break;

        /* buddy 在 free_list[order] 中，移除 buddy */
        list_del(&(b->page_link));
```

```
        /* 清除 buddy 的 property 标记 */
        ClearPageProperty(b);
        b->property = 0;

        /* 选择低地址作为新的 head */
        if (b < head) head = b;

        /* 合并后阶数+1 */
        ++order;
    }

    /* 将合并后的块加入 free_list[order] */
    add_block_to_freelist(head, order);
}
```

首先确认传入 `n` 必须是 2^{order}。这与 `alloc_pages` 的返回保证一致（该实现假定分配器给出的块大小是指数）。

清除 `PG_reserved`（页面现在可用）、引用计数置 0，并清楚 `PG_property`。

将 `head = base` 作为当前块头，进入合并循环：

- 计算伙伴 `b = buddy_of(head, order)`。
- 若 `b` 是空闲头（`PageProperty(b)` 为真）且其大小匹配（`b->property == 2^order`），则可以合并：
  - 从对应链表中删除 `b`（`list_del`），清除 `PG_property`。
  - 取低地址的那一半作为新的 `head`（`if (b < head) head = b;`）。
  - `order++`（合并后块大小翻倍）。
- 否则停止合并。

最后把合并后的大块以 `head` 形式加入 `free_lists[order]`。

这样释放后会尽可能向上合并，减少碎片。

检查与功能测试

1. **基础一致性检查**

```
size_t total = 0;
    for (unsigned int o = 0; o < BUDDY_MAX_ORDER; ++o) {
        list_entry_t *le = &free_lists[o];
        while ((le = list_next(le)) != &free_lists[o]) {
            struct Page *p = le2page(le, page_link);
            assert(PageProperty(p));
            total += p->property;
        }
    }
    assert(total == free_count_pages);
```

   - **目的：** 检查所有空闲链表的总页数是否正确。
   - **做法：**
     1. 遍历每个阶的 `free_lists[o]` 双向链表。

2. `le2page(le, page_link)` 获取每个空闲块的 `Page` 头信息。
3. `assert(PageProperty(p))` 确认链表里都是块头页。
4. 累加每个块的页数 `p->property` 到 `total`。
5. 最后 `assert(total == free_count_pages)` 检查统计的总页数是否和 buddy 管理器记录的空闲页数一致。

这部分保证了空闲链表的数据结构和总页数与实际管理的一致。

2. **功能测试**

通过**分配和释放不同大小的内存块**，模拟实际操作来自动化检查 buddy 算法是否正确。

分配1页：

```
struct Page *p1 = alloc_pages(1);
    if (!p1) {
        cprintf("[buddy_check] ERROR: alloc_pages(1) returned NULL\n");
        panic("buddy_check failed: alloc 1 page");
    }
    if (nr_free_pages() != before - 1) {
        cprintf("[buddy_check] WARNING: nr_free_pages before=%zu
after_alloc1=%zu\n",
                before, nr_free_pages());
        /* 继续，但记录问题 */
    }
    cprintf("[buddy_check] alloc 1 page OK\n");
```

检查：

- 分配 1 页是否成功。
- 分配后空闲页数是否减少 1。

释放1页：

```
 free_pages(p1, 1);
    if (nr_free_pages() != before) {
        cprintf("[buddy_check] WARNING: nr_free_pages after free1 expected=%zu
actual=%zu\n",
                before, nr_free_pages());
    }
    cprintf("[buddy_check] free 1 page OK\n");
```

检查：

- 释放后空闲页数是否恢复。

分配/释放8页：

```
struct Page *p8 = alloc_pages(8);
    if (!p8) {
        cprintf("[buddy_check] ERROR: alloc_pages(8) returned NULL\n");
        panic("buddy_check failed: alloc 8 pages");
    }
    if (nr_free_pages() != before - 8) {
```

```
        cprintf("[buddy_check] WARNING: nr_free_pages after alloc8 expected=%zu
actual=%zu\n",
                (size_t)(before - 8), nr_free_pages());
    }
    cprintf("[buddy_check] alloc 8 pages OK\n");

    free_pages(p8, 8);
    if (nr_free_pages() != before) {
        cprintf("[buddy_check] WARNING: nr_free_pages after free8 expected=%zu
actual=%zu\n",
                before, nr_free_pages());
    }
    cprintf("[buddy_check] free 8 pages OK\n");
```

检查：

- 分配多页块（$2^3 = 8$ 页）是否成功。
- 多页释放后空闲页数是否恢复。

最核心：合并测试：

```
struct Page *a = alloc_pages(8);
    if (!a) {
        cprintf("[buddy_check] ERROR: alloc_pages(8) for a returned NULL\n");
        panic("buddy_check failed: alloc a");
    }
    struct Page *b = alloc_pages(8);
    if (!b) {
        cprintf("[buddy_check] ERROR: alloc_pages(8) for b returned NULL\n");
        panic("buddy_check failed: alloc b");
    }

    free_pages(a, 8);
    free_pages(b, 8);

    /* 尝试申请 16 页，应当成功（如果合并正确） */
    struct Page *c = alloc_pages(16);
    if (!c) {
        cprintf("[buddy_check] ERROR: alloc_pages(16) returned NULL (merge
failed?)\n");
        panic("buddy_check failed: alloc 16");
    }
    free_pages(c, 16);
    cprintf("[buddy_check] merge test passed\n");
```

检查：

- 分配两块同阶的连续空闲页（8 页）后释放。
- 再申请 16 页（2^4），要求 buddy 能自动合并两块 8 页为 16 页。如果申请成功，则说明合并正确。

之后定义接口`buddy_pmm_manager`，在`grade.sh`文件中增加：

```
pts=5
quick_check 'check_buddy_free_count' \
    '[buddy_check] alloc 1 page OK' \
    '[buddy_check] free 1 page OK' \
    '[buddy_check] alloc 8 pages OK' \
    '[buddy_check] free 8 pages OK' \
    '[buddy_check] merge test passed'
```

通过匹配我们测试的字符串输出来完成判断。

测试结果

```
uki@uki-virtual-machine:~/riscv/labcode/lab2$ make grade
>>>>>>>>>> here_make>>>>>>>>>>>
gmake[1]: Entering directory '/home/uki/riscv/labcode/lab2' + cc kern/init/entry
.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/panic.c + cc ker
n/driver/console.c + cc kern/driver/dtb.c + cc kern/mm/best_fit_pmm.c + cc kern/
mm/buddy_system.c + cc kern/mm/default_pmm.c + cc kern/mm/pmm.c + cc libs/printf
mt.c + cc libs/readline.c + cc libs/sbi.c + cc libs/string.c + ld bin/kernel ris
cv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img gmake[1]
: Leaving directory '/home/uki/riscv/labcode/lab2'
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=10683
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_buddy:                              OK
  -check_buddy_free_count:                   OK
Total Score: 30/30
```

可以看到得到了测试分数。