Buddy System (伙伴系统) 分配算法设计文档

概述

Buddy System 是一种内存分配算法，通过将内存块划分为大小为 2 的幂次方的块来管理内存。它通过合并和分割内存块来高效地分配和释放内存。Buddy System 的主要优点是能够快速地找到合适大小的空闲块，并且在释放内存时能够高效地合并相邻的空闲块。

结构体和宏定义

`free_area_t`

`free_area_t` 结构体用于表示每个大小类别的空闲内存块链表和空闲块数量。

```
typedef struct {
    list_entry_t free_list; // 空闲链表
    size_t nr_free;         // 空闲块数量
} free_area_t;
```

- `free_list`：一个链表，存储了当前大小类别的所有空闲内存块。
- `nr_free`：当前大小类别的空闲内存块数量。

宏定义

以下宏定义用于简化代码中的一些操作：

```
#define MAX_ORDER 11
#define free_list(i) free_buddy_area[(i)].free_list
#define nr_free(i) free_buddy_area[(i)].nr_free
#define IS_POWER_OF_2(x) (!((x)&((x)-1)))
```

- `MAX_ORDER`：定义了最大的块大小类别。
- `free_list(i)` 和 `nr_free(i)`：用于访问 `free_buddy_area` 数组中的元素。
- `IS_POWER_OF_2(x)`：用于判断一个数是否是 2 的幂。

函数实现

初始化函数

`buddy_system_init`

`buddy_system_init` 函数初始化 Buddy System 的空闲链表和空闲块数量。

```
static void buddy_system_init(void) {
    for(int i = 0; i < MAX_ORDER; i++) {
```

```
        list_init(&(free_buddy_area[i].free_list));
        free_buddy_area[i].nr_free = 0;
    }
}
```

该函数遍历所有的大小类别（从 0 到 `MAX_ORDER - 1`），并初始化每个类别的空闲链表和空闲块数量。

buddy_system_init_memmap

`buddy_system_init_memmap` 函数初始化内存映射，将物理内存块加入到相应的空闲链表中。

```
static void buddy_system_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    size_t curr_size = n;
    uint32_t order = MAX_ORDER - 1;
    uint32_t order_size = 1 << order;
    p = base;
    while (curr_size != 0) {
        p->property = order_size;
        SetPageProperty(p);
        nr_free(order) += 1;
        list_add_before(&(free_list(order)), &(p->page_link));
        curr_size -= order_size;
        while(order > 0 && curr_size < order_size) {
            order_size >>= 1;
            order -= 1;
        }
        p += order_size;
    }
}
```

该函数首先检查输入参数 `n` 是否大于 0。然后遍历从 `base` 开始的 `n` 个页面，确保每个页面都是保留状态，并重置其标志和引用计数。接下来，根据页面数量 `n` 初始化相应的空闲链表，将页面块加入到相应的空闲链表中。

分配和释放内存

buddy_system_alloc_pages

`buddy_system_alloc_pages` 函数分配指定数量的页面。

```

```

```
static struct Page *buddy_system_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > (1 << (MAX_ORDER - 1))) {
        return NULL;
    }
    struct Page *page = NULL;
    uint32_t order = MAX_ORDER - 1;
    while (n < (1 << order)) {
        order -= 1;
    }
    order += 1;
    uint32_t flag = 0;
    for (int i = order; i < MAX_ORDER; i++) flag += nr_free(i);
    if(flag == 0) return NULL;
    if(list_empty(&(free_list(order)))) {
        split_page(order + 1);
    }
    if(list_empty(&(free_list(order)))) return NULL;
    list_entry_t *le = list_next(&(free_list(order)));
    page = le2page(le, page_link);
    list_del(&(page->page_link));
    ClearPageProperty(page);
    return page;
}
```

该函数首先检查输入参数 `n` 是否大于 0，并且 `n` 是否小于等于最大块大小。然后根据 `n` 计算所需的块大小类别 `order`。接下来检查是否有足够的空闲块，如果没有则尝试分割更大的块。最后从空闲链表中取出一个块，并返回该块的页面指针。

buddy_system_free_pages

`buddy_system_free_pages` 函数释放指定数量的页面。

```
static void buddy_system_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(IS_POWER_OF_2(n));
    assert(n < (1 << (MAX_ORDER - 1)));
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);

    uint32_t order = 0;
    size_t temp = n;
    while (temp != 1) {
        temp >>= 1;
```

```
            order++;
        }
        add_page(order, base);
        merge_page(order, base);
    }
```

该函数首先检查输入参数 `n` 是否大于 0，是否是 2 的幂，并且 `n` 是否小于最大块大小。然后遍历从 `base` 开始的 `n` 个页面，确保每个页面都没有被保留且没有属性标志，并重置其标志和引用计数。接下来根据 `n` 计算块大小类别 `order`，并将块加入到相应的空闲链表中。最后尝试合并相邻的空闲块。

辅助函数

`split_page`

`split_page` 函数将高一级的空闲块分割为两个较小的块。

```
    static void split_page(int order) {
        if(list_empty(&(free_list(order)))) {
            split_page(order + 1);
        }
        list_entry_t* le = list_next(&(free_list(order)));
        struct Page *page = le2page(le, page_link);
        list_del(&(page->page_link));
        nr_free(order) -= 1;
        uint32_t n = 1 << (order - 1);
        struct Page *p = page + n;
        page->property = n;
        p->property = n;
        SetPageProperty(p);
        list_add(&(free_list(order-1)),&(page->page_link));
        list_add(&(page->page_link),&(p->page_link));
        nr_free(order-1) += 2;
        return;
    }
```

该函数首先检查当前大小类别的空闲链表是否为空，如果为空则递归调用自身以分割更大的块。然后从空闲链表中取出一个块，将其分割为两个较小的块，并将这两个较小的块加入到较低一级的空闲链表中。

`add_page`

`add_page` 函数将块按照地址从小到大的顺序加入到指定序号的链表中。

```
    static void add_page(uint32_t order, struct Page* base) {
        if (list_empty(&(free_list(order)))) {
            list_add(&(free_list(order)), &(base->page_link));
        } else {
            list_entry_t* le = &(free_list(order));
            while ((le = list_next(le)) != &(free_list(order))) {
```

```
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &(free_list(order))) {
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

该函数首先检查当前大小类别的空闲链表是否为空，如果为空则直接将块加入到链表中。如果链表不为空，则遍历链表找到合适的位置，将块按照地址从小到大的顺序加入到链表中。

`merge_page`

`merge_page` 函数合并相邻的空闲块。

```
static void merge_page(uint32_t order, struct Page* base) {
    if (order == MAX_ORDER - 1) {
        return;
    }

    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &(free_list(order))) {
        struct Page *p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
            if(order != MAX_ORDER - 1) {
                list_del(&(base->page_link));
                add_page(order+1,base);
            }
        }
    }

    le = list_next(&(base->page_link));
    if (le != &(free_list(order))) {
        struct Page *p = le2page(le, page_link);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
            if(order != MAX_ORDER - 1) {
                list_del(&(base->page_link));
                add_page(order+1,base);
            }
        }
    }
```

```
        merge_page(order+1,base);
        return;
    }
```

该函数首先检查当前块是否已经是最大块，如果是则直接返回。然后检查当前块的前一个块是否与当前块相邻，如果相邻则合并这两个块，并将合并后的块加入到更高一级的空闲链表中。接下来检查当前块的后一个块是否与当前块相邻，如果相邻则合并这两个块，并将合并后的块加入到更高一级的空闲链表中。最后递归调用自身以尝试合并更高一级的块。

检查函数

`buddy_system_check`

`buddy_system_check` 函数检查 Buddy System 的正确性。

```
static void buddy_system_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(p0 != p1 && p0 != p2 && p1 != p2);
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);

    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);
    for(int i = 0; i < MAX_ORDER; i++) {
        list_init(&(free_list(i)));
        assert(list_empty(&(free_list(i))));
    }

    for(int i = 0; i < MAX_ORDER; i++) {
        list_init(&(free_list(i)));
        assert(list_empty(&(free_list(i))));
    }
    for(int i = 0; i < MAX_ORDER; i++) nr_free(i) = 0;

    assert(alloc_page() == NULL);

    free_page(p0);
    free_page(p1);
    free_page(p2);
    assert(buddy_system_nr_free_pages() == 3);

    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(alloc_page() == NULL);
```

```
    free_page(p0);
    for(int i = 0; i < 0; i++) assert(!list_empty(&(free_list(i))));

    struct Page *p;
    assert((p = alloc_page()) == p0);
    assert(alloc_page() == NULL);

    assert(buddy_system_nr_free_pages() == 0);

    free_page(p);
    free_page(p1);
    free_page(p2);
}
```

该函数首先分配三个页面，并确保它们是不同的页面且引用计数为 0。然后检查这些页面的物理地址是否在有效范围内。接下来初始化所有大小类别的空闲链表，并确保它们为空。然后释放之前分配的三个页面，并确保空闲页面数量为 3。接下来再次分配三个页面，并确保分配成功。最后释放所有页面，并确保空闲页面数量为 0。

统计函数

`buddy_system_nr_free_pages`

`buddy_system_nr_free_pages` 函数计算空闲页面的数量。

```
static size_t buddy_system_nr_free_pages(void) {
    size_t num = 0;
    for(int i = 0; i < MAX_ORDER; i++) {
        num += nr_free(i) << i;
    }
    return num;
}
```

该函数遍历所有大小类别的空闲链表，计算每个类别的空闲页面数量，并返回总的空闲页面数量。

`pmm_manager` 结构体

`pmm_manager` 结构体定义了 Buddy System 内存管理器的接口。

```
const struct pmm_manager buddy_system_pmm_manager = {
    .name = "default_pmm_manager",
    .init = buddy_system_init,
    .init_memmap = buddy_system_init_memmap,
    .alloc_pages = buddy_system_alloc_pages,
    .free_pages = buddy_system_free_pages,
    .nr_free_pages = buddy_system_nr_free_pages,
```

```
    .check = buddy_system_check,
};
```

- `name`：内存管理器的名称。
- `init`：初始化函数。
- `init_memmap`：初始化内存映射函数。
- `alloc_pages`：分配页面函数。
- `free_pages`：释放页面函数。
- `nr_free_pages`：计算空闲页面数量函数。
- `check`：检查函数。

总结

Buddy System 内存管理通过分割和合并内存块来高效地管理内存。它使用空闲链表来跟踪不同大小的内存块，并提供了分配和释放内存的功能。通过检查函数可以验证 Buddy System 的正确性。Buddy System 的主要优点是能够快速地找到合适大小的空闲块，并且在释放内存时能够高效地合并相邻的空闲块。