扩展练习Challenge：buddy system（伙伴系统）分配算法（需要编程）

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

- *参考伙伴分配器的一个极简实现， 在ucore中实现buddy system分配算法，要求有比较充分的测试用例说明实现的正确性，需要有设计文档。*

一、概述

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理，每个存储块的大小必须是2的n次幂(Pow(2, n))，即1, 2, 4, 8, 16, 32, 64, 128...

在实现伙伴系统分配算法之前，首先回顾一下伙伴系统的基本概念。伙伴系统分配算法通过将内存按2的幂进行划分，以便于高效管理内存的分配与释放。该算法利用空闲链表来维护不同大小的空闲内存块，能够快速查找合适大小的块，且在合并时能够减少外部碎片。尽管其优点明显，但也存在内部碎片的问题，尤其在请求大小不是2的幂次方时。

其实我们解决的主要问题就是分配和收回，其他我们的设计基本上和first-fit差不多，但是我们需要解决的是，我们使用什么样子的数据结构来完成内存块的分配，在参考文件中，提到了一种二叉树的数据结构的实现方法，但是实际上，该方法虽然很简便，但是有一定的缺点，就是**内存开销过于大**了。。。，所以，我们采用一种数组链表的实现方法。

除此之外，关于分配和收回的相关设计，将在接下来进行。

二、开发文档

1、设计数据结构

由于我们buddy_system的每一个存储块的大小必须是2的n次方的大小，所以我们需要重新设计新的数据结构，来存放我们对应的块，参考之前的设计，我们设计新的数据结构。

```
#define MAX_BUDDY_ORDER 15  // 支持最大块 2^15 页


static list_entry_t buddy_array[MAX_BUDDY_ORDER + 1];  // 每阶的空闲链表（双向链表）
static size_t nr_free;                                 // 当前空闲页总数
extern struct Page *pages;                             // 全局页数组
extern size_t npage;
#define page2idx(p) ((p) - pages)
#define idx2page(i) (pages + (i))
```

实际上我们需要分配的是31929个页，我们采用了首先利用数组来表示每层块（所谓每层就是指2的k次页数的块），也就是**buddy_array[k]** 管理系统中所有当前“大小为 2^k 页”的空闲块，每个空闲块利用每一个链表将所有的空闲块接起来；然后nr_free表示所有空闲页的总和。

其实实际上我们分配的是31929个页，如果初始分配的话，我们可以见到的是这一定会分配成一个大块和很多小块。

2、初始化设计

把 `[base, base+n)` 区间**以尽可能大的 2^k 对齐块**划分（对齐到页号上）；插入 `buddy_array[k]` 对应链表，使链表中的块互不重叠并且覆盖所有可用页。；更新 `nr_free` 为所有页之和；使得若 `[base, base+n)` 自身对齐且大小恰为 `2^t`，则仅在 `buddy_array[t]` 中插入一个块（理想情况）。

对某个起始页号 `addr`，要将其作为大小 `2^k` 的块，需要满足：

- `2^k ≤ n`（块不能超过剩余页数）
- `addr % 2^k == 0`（页号对齐）
  满足这两个条件的最大 `k` 就是应选的阶。

```
addr = page2idx(base); remaining = n;
while remaining > 0:
    // 找最大的 block_size = 2^k <= remaining 且 addr % block_size == 0
    k = floor(log2(largest_power_of_two <= remaining and aligned))
    add block [addr, addr + 2^k - 1] into buddy_array[k]
    nr_free += 2^k
    addr += 2^k
    remaining -= 2^k
```

3、分配机制

首先，先说明分配机制的核心：从上往下找合适的块；如果太大，就不断二分拆小；直到得到刚好能容纳请求的块。

```
/* 分配 pages：把请求上调为 2^k，然后找合适块并分裂直至到达 k */
static struct Page *buddy_system_alloc_pages(size_t requested_pages) {
    assert(requested_pages > 0);
    if (requested_pages > nr_free) return NULL;

    size_t size = round_up_pow2(requested_pages);
    unsigned int order = order_of_pow2(size);
    if (order > MAX_BUDDY_ORDER) return NULL;

    /* 找到 >= order 的第一个非空链表 */
    unsigned int i = order;
    while (i <= MAX_BUDDY_ORDER && list_empty(&buddy_array[i])) i++;
    if (i > MAX_BUDDY_ORDER) return NULL;

    /* 自上而下分裂直到 order */
    while (i > order) {
        /* split the first block in buddy_array[i] into two of order i-1 */
        buddy_system_split(i);
        i--;
    }

    /* 现在 buddy_array[order] 非空，拿出第一个块 */
    list_entry_t *le = list_next(&buddy_array[order]);
    struct Page *page = le2page(le, page_link);
    list_del(le);
    ClearPageProperty(page);
```

```
    /* 更新总空闲页 */
    nr_free -= (1u << order);
    return page;
}
```

当系统收到一个内存页分配请求时，它首先把请求页数向上取整为最接近的 2 的幂（保证块大小合法），然后确定对应的阶次 `order`。接着，它从这一阶开始查找空闲块链表 `buddy_array[order]`，若当前阶次没有可用块，就向更高阶（更大的块）寻找。当找到一个足够大的块后，如果块比需要的大，就从高阶往低阶不断将其对半分裂，每次分裂出的另一半重新放入相应阶次的空闲链表中。最终，当块大小恰好满足请求时，从对应链表中取出一个块首页、标记为已分配，并更新系统的空闲页计数 `nr_free`。

4、释放机制

首先，释放机制的核心就是：从下往上找不断找到相同的块进行合并。

```
/* 释放 pages：base 必须是块头页（property 指示阶）或者 n 给出释放页数与块一致 */
static void buddy_system_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    /* 将 n 调整为块大小（必须是 2^k），并计算order */
    size_t blk_size = round_up_pow2(n);
    unsigned int order = order_of_pow2(blk_size);
    assert((1u << order) == blk_size);

    /* 将 base 标记为该 order 的块头并插入 */
    base->property = order;
    SetPageProperty(base);
    list_add(&buddy_array[order], &base->page_link);

    /* 尝试合并 */
    struct Page *left = base;
    while (order < MAX_BUDDY_ORDER) {
        struct Page *buddy = get_buddy(left, order);
        if (buddy == NULL) break;
        /* 只有当伙伴存在且是空闲且同阶时才合并 */
        if (!PageProperty(buddy) || buddy->property != order) break;
        /* 从链表中删除 left 和 buddy（注意可能 buddy 在链表任何位置） */
        list_del(&left->page_link);
        list_del(&buddy->page_link);
        /* 计算合并后新的左侧基址（较小的地址）*/
        if (left > buddy) {
            struct Page *tmp = left;
            left = buddy;
            buddy = tmp;
        }
        /* 新块阶升一 */
        order++;
        left->property = order;
        /* 将新的大块插回对应链表以便可能继续合并 */
        list_add(&buddy_array[order], &left->page_link);
    }
```

当系统释放一个内存块时，它首先把该块标记为空闲并插入对应阶的链表，然后计算其伙伴块位置。如果发现伙伴也空闲且大小相同，就将两者从链表中删除并合并为一个更大阶的块，再把新块插入更高阶链表中继续尝试合并。这个自下而上的合并过程会持续进行，直到伙伴不存在或已被使用。最终，所有相邻同阶的空闲块都会自动拼接成更大的连续块，从而有效减少外部碎片并维持内存的层次平衡结构。

三、测试样例

关于测试，我们需要一个合理的测试案例用来表示我们的结果是正确的。ok，首先的是一个初始化的测试。

我们先展示我们的测试代码。

```
static void buddy_system_check_easy_alloc_and_free_condition(void) {
    cprintf("CHECK OUR EASY ALLOC CONDITION:\n");
    cprintf("当前总的空闲页的数量为：%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;

    cprintf("首先,p0请求16页\n");
    p0 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("然后,p1请求4页\n");
    p1 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("最后,p2请求500页\n");
    p2 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("p0 idx=%u\n", (unsigned)page2idx(p0));
    cprintf("p1 idx=%u\n", (unsigned)page2idx(p1));
    cprintf("p2 idx=%u\n", (unsigned)page2idx(p2));

    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);

    cprintf("CHECK OUR EASY FREE CONDITION:\n");
    cprintf("释放p0...\n");
    free_pages(p0, 10);
    cprintf("释放p0后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p1...\n");
    free_pages(p1, 10);
    cprintf("释放p1后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p2...\n");
```

```
    free_pages(p2, 10);
    cprintf("释放p2后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}



static void buddy_system_check_min_alloc_and_free_condition(void) {
    struct Page *p3 = alloc_pages(1);
    cprintf("分配p3之后(1页)\n");
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, 1);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}

static void buddy_system_check_max_alloc_and_free_condition(void) {
    size_t big = 1u << MAX_BUDDY_ORDER;
    if (big > npage) big = round_down_pow2(npage);
    struct Page *p3 = alloc_pages(big);
    cprintf("分配p3之后(%u页)\n", (unsigned)big);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, big);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}

static void buddy_system_check(void) {
    cprintf("BEGIN TO TEST OUR BUDDY SYSTEM!\n");
    buddy_system_check_easy_alloc_and_free_condition();
    buddy_system_check_min_alloc_and_free_condition();
    buddy_system_check_max_alloc_and_free_condition();
}
```

1、首先我们先进行的是初始化的测试。

```
CHECK OUR EASY ALLOC CONDITION:
当前总的空闲页的数量为：31929
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】
```

```
order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

我们的内存的总页数应该是31929页，在buddy_system内存分配的算法下，我们首先做的是尽可能将空闲页数结合成一个大块（也就是2的k次方页大小的块），然后剩下的块再进行分配，结果我们就得到了相应的数组链表，也就是我们的空闲页数组链表，不难发现的是，所有空闲页加起来也就是我们相应的31929页空闲页，说明我们的初始化很正确。

2、请求和释放机制测试

我们的测试代码：

```
static void buddy_system_check_easy_alloc_and_free_condition(void) {
    cprintf("CHECK OUR EASY ALLOC CONDITION:\n");
    cprintf("当前总的空闲页的数量为：%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;

    cprintf("首先,p0请求16页\n");
    p0 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("然后,p1请求4页\n");
    p1 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("最后,p2请求500页\n");
    p2 = alloc_pages(10);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("p0 idx=%u\n", (unsigned)page2idx(p0));
    cprintf("p1 idx=%u\n", (unsigned)page2idx(p1));
    cprintf("p2 idx=%u\n", (unsigned)page2idx(p2));

    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);

    cprintf("CHECK OUR EASY FREE CONDITION:\n");
    cprintf("释放p0...\n");
    free_pages(p0, 10);
    cprintf("释放p0后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    cprintf("释放p1...\n");
    free_pages(p1, 10);
    cprintf("释放p1后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
```

```
    cprintf("释放p2...\n");
    free_pages(p2, 10);
    cprintf("释放p2后,总空闲页数为:%u\n", (unsigned)nr_free);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}
```

首先我们先申请了一个16页，在初始化的链表展示中，我们发现我们有一个16页的块，所以，直接将这个块分配，结果就是这个块不再是空闲块。

如下所示，和初始化的相比。我们失去了16页的那个块，实际上也就是分配掉了。说明分配很成功。

```
首先,p0请求16页
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下来我们申请一个4页的空间，由于我们没有一个正好4页的块，所以必须要进行分裂，正好我们有一个8页的空闲块，将它分裂成两个4页的空闲块，然后进行分配，就可以了。

如下所示，我们确实将那个8页的块分裂了，然后其中一块分配掉了，剩下的一块成为了空闲的。

```
然后,p1请求4页
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  2 : [4 pages, idx=840] 【地址为0xffffffffc020f340】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】
```

```
order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下里，我们申请一个500页的空间，但是，由于我们的块都是2的k次方的页，所以距离400页最近的空闲块将被分配，也就是512页的块，上次分配后的空闲块，我们发现有一个1024页的空闲块，所以我们要将这个块进行分裂，然后将其中一个512页的块进行分配。

如下图所示，我们确实将那个1024页的空闲块分裂，其中一个分配掉了。所以只剩下一个512页的空闲块。

```
最后,p2请求500页
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  2 : [4 pages, idx=840] 【地址为0xffffffffc020f340】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order  9 : [512 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下来就是我们的收回机制。

首先收回p0也就是16页的那个块。发现，不用合并。

```
释放p0...
释放p0后,总空闲页数为:31413
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  2 : [4 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order  9 : [512 pages, idx=1024] 【地址为0xffffffffc0211000】
```

```
order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下来我们收回那个4页的块p1，结果应该会和空闲块合并成一个8页的空闲块。当然我们的总空闲块页数也变成了31417页。

```
释放p1...
释放p1后,总空闲页数为:31417
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order  9 : [512 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

接下来我们收回那个500页的块，实际上os给他分配了512页的块，结果显而易见，就是要合并成一个1024页的空闲块。

```
释放p2...
释放p2后,总空闲页数为:31929
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】
```

```
order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

3、测试最小请求和释放

```
static void buddy_system_check_min_alloc_and_free_condition(void) {
    struct Page *p3 = alloc_pages(1);
    cprintf("分配p3之后(1页)\n");
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, 1);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}
```

结果如下：实际就是分配了那个一页的块。

```
分配p3之后(1页)
------------------当前空闲的链表数组:------------------
order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```

4、测试最大请求和释放

```
static void buddy_system_check_max_alloc_and_free_condition(void) {
    size_t big = 1u << MAX_BUDDY_ORDER;
    if (big > npage) big = round_down_pow2(npage);
    struct Page *p3 = alloc_pages(big);
    cprintf("分配p3之后(%u页)\n", (unsigned)big);
    show_buddy_array(0, MAX_BUDDY_ORDER);

    free_pages(p3, big);
    show_buddy_array(0, MAX_BUDDY_ORDER);
}
```

这种显然是请求会失败，因为确实没有这么大的块。实际上最大请求是16384页这么大的块

结果如下：

```
分配p3之后(32768页)
------------------当前空闲的链表数组:------------------
order  0 : [1 pages, idx=839] 【地址为0xffffffffc020f318】

order  3 : [8 pages, idx=840] 【地址为0xffffffffc020f340】

order  4 : [16 pages, idx=848] 【地址为0xffffffffc020f480】

order  5 : [32 pages, idx=864] 【地址为0xffffffffc020f700】

order  7 : [128 pages, idx=896] 【地址为0xffffffffc020fc00】

order 10 : [1024 pages, idx=1024] 【地址为0xffffffffc0211000】

order 11 : [2048 pages, idx=2048] 【地址为0xffffffffc021b000】

order 12 : [4096 pages, idx=4096] 【地址为0xffffffffc022f000】

order 13 : [8192 pages, idx=8192] 【地址为0xffffffffc0257000】

order 14 : [16384 pages, idx=16384] 【地址为0xffffffffc02a7000】

------------------显示完成!------------------
```