

# 伙伴系统物理内存分配器

注: 所有来自互联网的图片会被做特殊标注, 其余图片为组员绘制。

YuanShen OS使用伙伴系统的物理内存管理方式, 伙伴系统的基本思想是将物理内存划分成连续的块, 以块为基本单位进行分配, 不同块的大小可以不同, 每个块都由一个或多个连续的物理页组成, 块的大小要求为2的幂。

## 1 传统设计

伙伴系统初始化在一段连续的物理内存区间上。根据Linux的精神, 伙伴系统是最底层的内存分配器, 为所有上层的次级分配器提供大块连续内存的补给, 在我们的下一个文档SLUB分配器中我们会再次提到这一点。

我们结合伙伴系统的传统设计来讲解基本伙伴系统的原理, 这块会快速带过, 之后我们会着重讲YuanShen OS中的独特设计。传统的伙伴系统使用多级链表组织维护内部未分配的物理页,如下图所示：

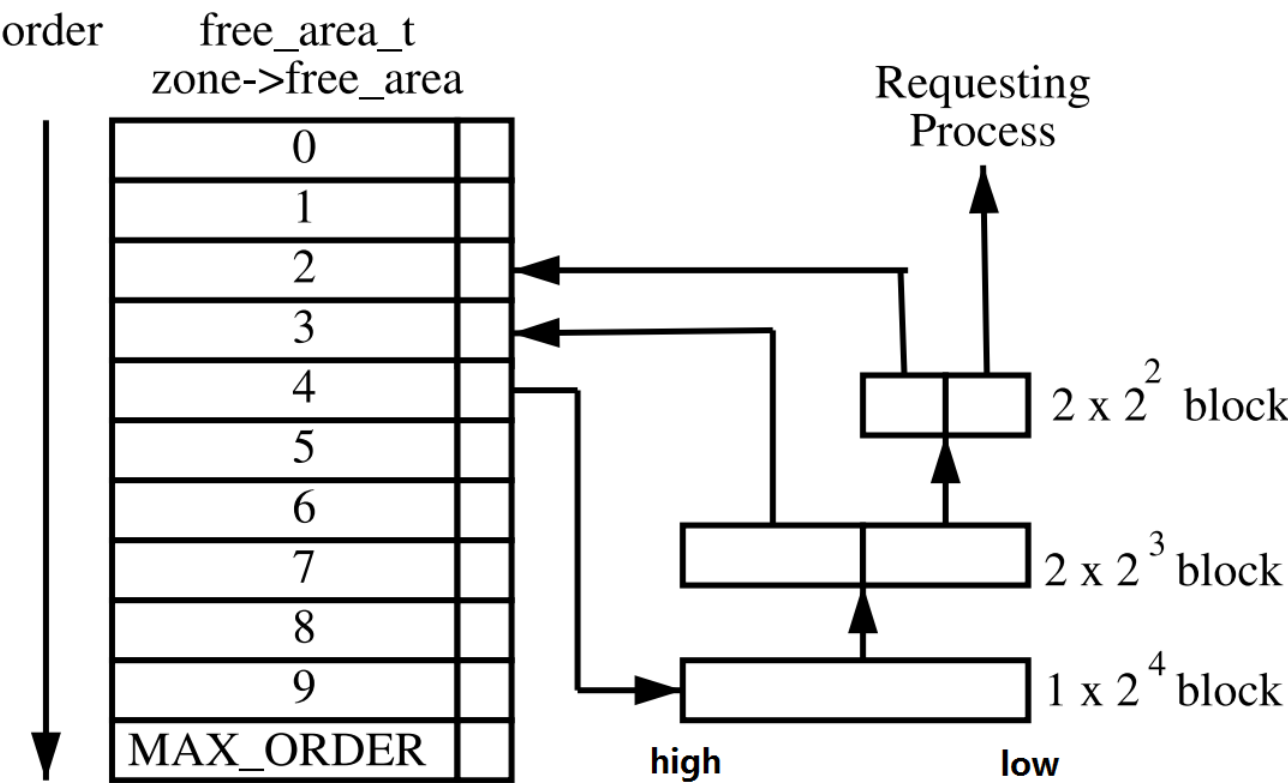


Fig1. 伙伴系统的传统设计(网图)

### 1.1 分配

最开始, 伙伴系统管理的所有物理内存, 一个连续的长度为2的幂的区间, 由最下方`MAX_ORDER`表项对应的 `free list`维护。这个 `free list`里只有一个节点, 这一个节点就维护了整条区间。

当我们要分配若干页的时候, 伙伴系统会对表`zone->free_area`进行自顶向下的查找, 每一个表项对应的物理页区间长度不同。如图所示, 越往下区间长度越大, 且都是2的幂。直到我们找到某个表项满足

- 1. 对应的区间比request的页面数大。

2. 对应的free list非空

那么我们会把那个表项对应的free list上拿出一个区间。假设这个区间大小为16,但request的页面只有3，那么伙伴系统会把这个长为16页的区间打碎为两个长度为8的区间，插入上一级表项的free list中，这两个区间就被称作是一对"伙伴"，因为它们来源于同一个父区间。

但是8对于3而言还是太大了，因此我们会继续从大小8页对应的free list上取出一个刚刚放进去的区间，将它拆分为两个大小为4页的区间，再放进大小4对应的free list，如图所示。最后我们会直接把大小为4的区间从free list上再拿下来，然后用于最终的分配。

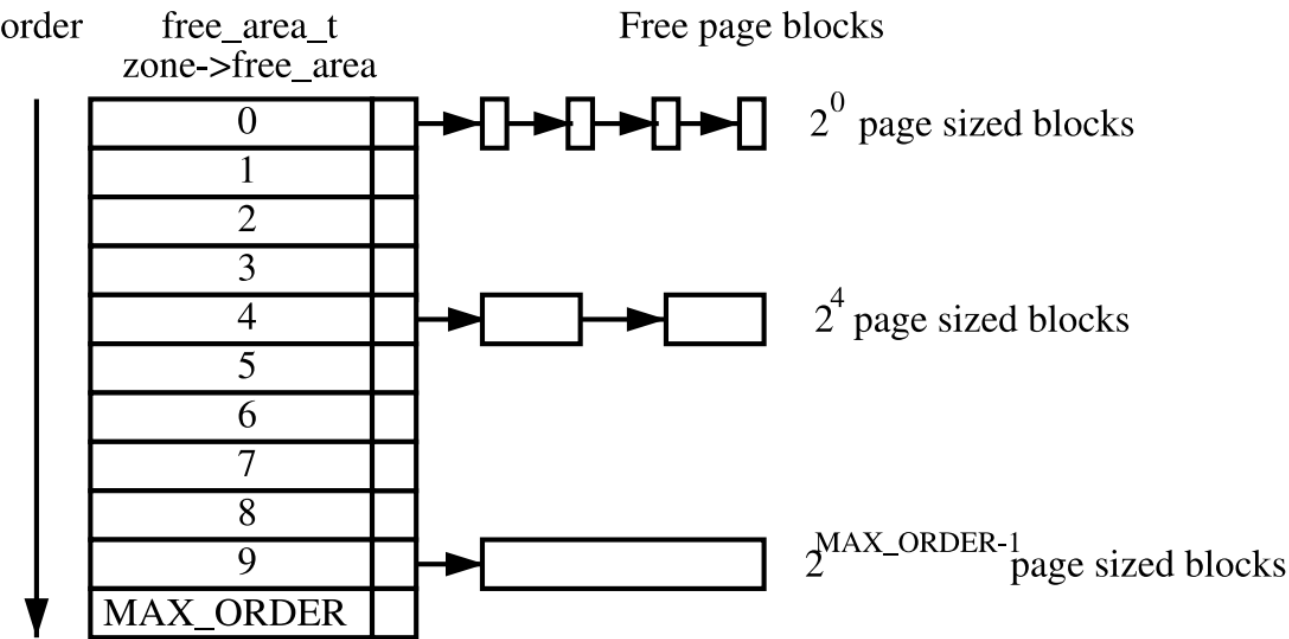


Fig2. 伙伴系统的free list设计(网图)

#### 回收性能瓶颈 伙伴系统的回收机制是伙伴系统的性能瓶颈。为了解决外部碎片问题，伙伴系统在回收一个连续区间后不只是简单的把这个区间插回对应的free list，还要检查这个free list内有没有它的伙伴。

如果有的话，那么就要把它们合并成一个更大的区间，放进下一级的free list内，然后重复上面这个过程。

2 基于完全二叉树的回收优化

查找free list中的伙伴并非易事，如果采取传统的单纯的多级链表的设计，这一步将是O(n)的时间复杂度。同时，传统设计存在的另一点不足是无法用简单的逻辑计算回收的地址从属于哪个free list。

YuanShen OS给出的方案利用了一个巧妙的事实，伙伴系统分配出来的区间可以看作是对一个连续区间不断二分法而得到的，因此所有可以被分配的区间可以用完全二叉树的对应结点描述。

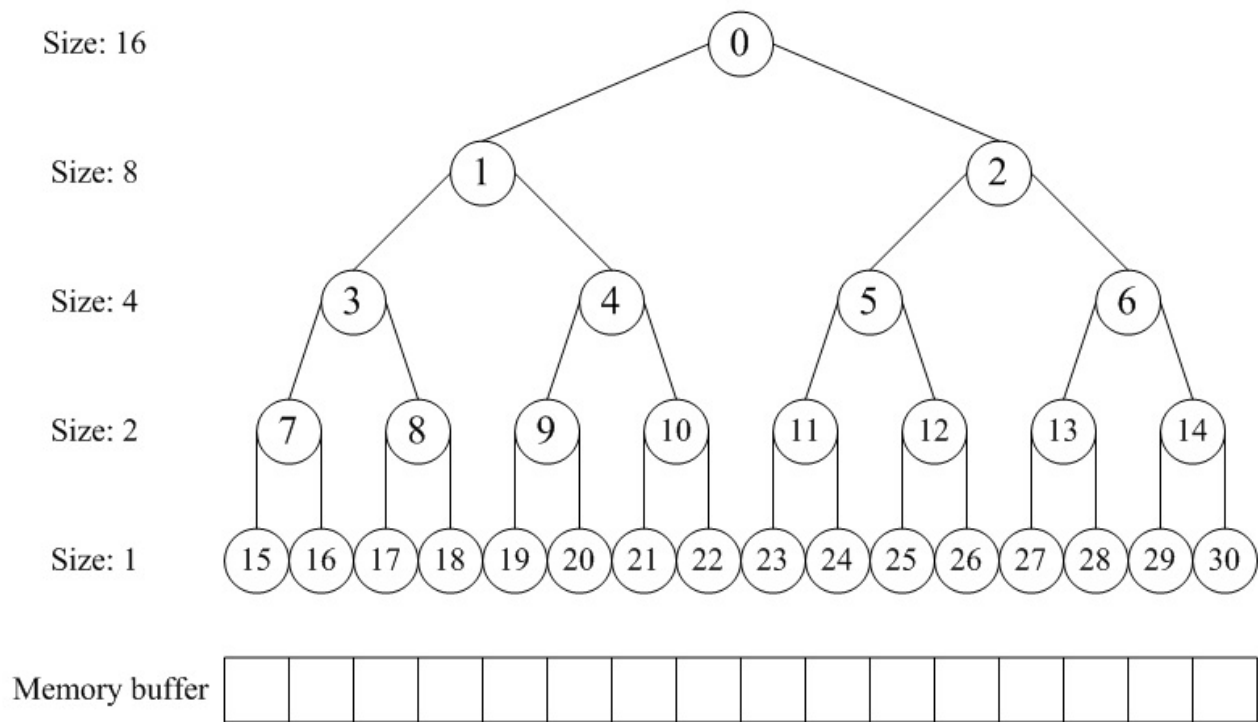


Fig3. 伙伴系统与完全二叉树的联系(网图)

在struct buddy\_中可以看到有个tree nodes的数组，它就是这个完全二叉树。同时，我们在伙伴系统里保留了freelist的设计，YuanShen OS的伙伴系统的设计的最大的特色在于：

1. 使用完全二叉树实现 $O(1)$ 时间复杂度判断区间的伙伴节点是否处在free list中，实现合并加速。
2. 同时为了减少开销，我们直接在二叉树的同一高度的结点间建立free list，同一层维护相同大小的连续区间。而使用数组实现二叉树的结构，可以让我们很方便的在父结点和兄弟结点之间移动。
3. 使用二叉树可以极大地简化确定待回收的区间所属的结点的高度的逻辑。

```
struct buddy_ {
    struct spinlock lock;
    uint64 base_addr;
    struct tree_node nodes[(1 << (LAYER_NUM + 1)) - 1];
    struct tree_node *freelists[LAYER_NUM];
};
```

从二叉树结点的类型声明上我们可以很容易观察到刚才陈述的结构，每个结点具有prev和next字段，用于维护同一高度的结点之间的free list。state字段是枚举类型，用于记录结点对应的状态，之后我们会用精简的例子说明YuanShen OS伙伴系统的工作原理。二叉树兼链表节点的代码如下所示：

```
struct tree_node {
    uint32 id;
    uint8 state;
    struct tree_node *next, *prev;
};
```

```
// 结点可能处于的状态
enum {UNUSED, ALLO, SPLIT, INLIST};
```

3 YuanShen OS的分配算法

YuanShen OS的伙伴系统设计在加入完全二叉树后，除了要在二叉树的结点上维护区间的状态信息外，在分配算法上没有什么别的变化。如前文所言，状态信息的维护主要是用于进行回收优化。

用一个例子来体验节点状态的维护要比直接看代码要轻松的多,简单起见，我们不妨假设伙伴系统最开始维护一个长为8的区间，而request size为2,我们来看看应该怎么修改二叉树的结点状态。

最开始二叉树的根结点为INLIST(蓝色)状态，表示它处在free list中，可以被拆分或者分配出去。而其它的结点的状态都是UNUSED(灰色)，表示既不在free list内，也不是某个刚好被分配出去的区间(但可以是某个被分配出去的区间的子区间)，也没有被分裂过。

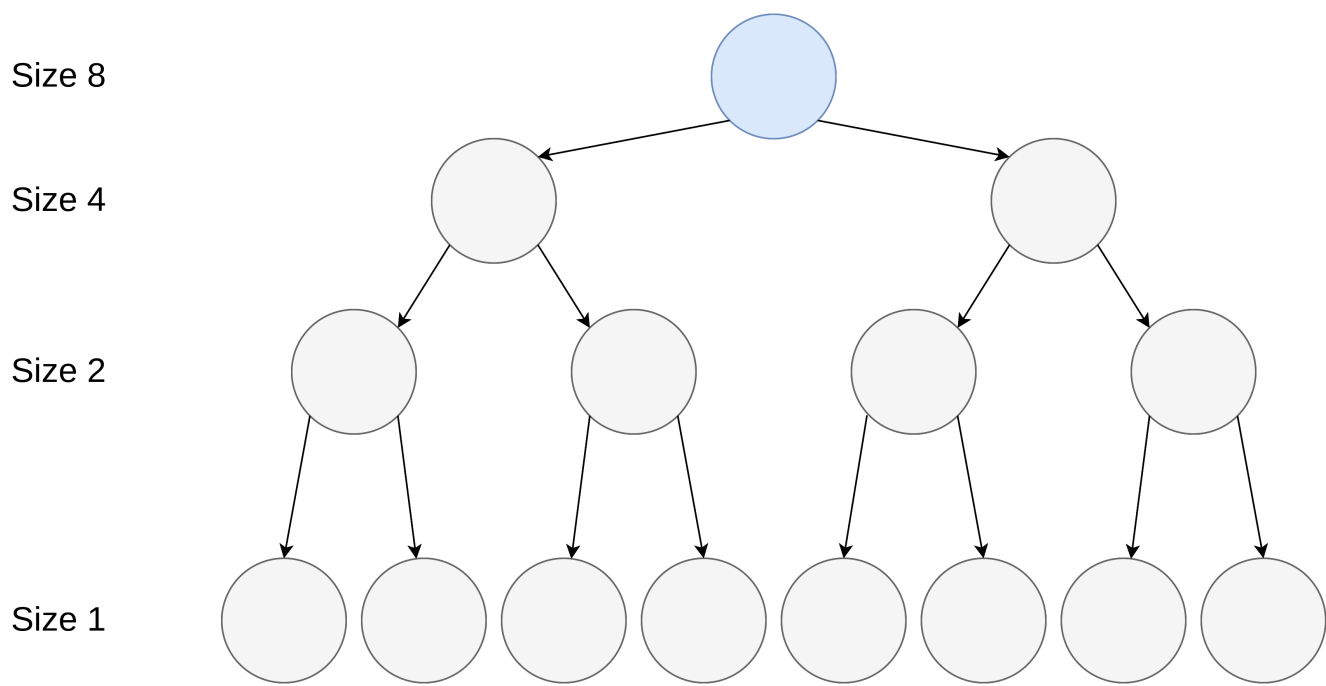


Fig4. 伙伴系统最初的状态(原创)

为了分配出大小为1的区间，我们自小到查询可用的free list，发现只有树根对应的一个长度为16的大区间，因此我们需要把这个大区间不断的拆分，并把拆分的产物放入对应size的free list：

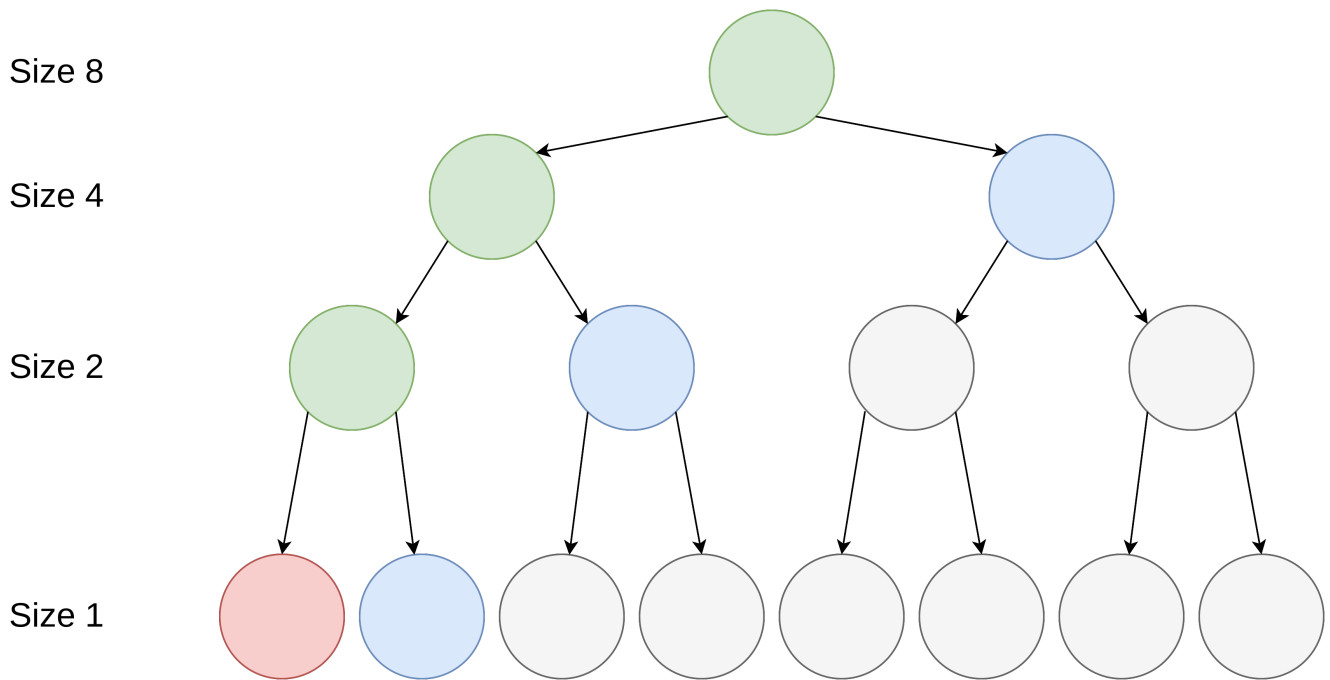


Fig5. 分配一个大小为1的区间(原创)

每次发生拆分的时候，结点的状态会被标记为SPLIT(绿色)，并把左右区间都标记为INLIST，也就是它的左右孩子结点，如果左区间要被进一步拆分，那就把左区间也标记成SPLIT，然后重复刚才的过程。最终被分配出去的区间对应的结点会标记成ALLO(红色)，值得注意的是它的子区间(如果有的话)不需要被标记为红色。

简化后的代码如下，主要展示二叉树维护的逻辑。

## 初始化

- $i \leftarrow 1$

## 步骤

### 1. 查找合适的层级:

- 重复以下操作，直到找到合适的层级或达到最大层级:
  - 如果  $2^{(i-1)} < \text{size}$  或  $\text{freelists}[\text{LAYER\_NUM} - i] == 0$ :
    - 如果  $i == \text{LAYER\_NUM}$ :
      - 返回 0 (分配失败)
    - 否则:
      - $i \leftarrow i + 1$

### 2. 处理分裂或直接分配:

- 如果  $i > 1$  且  $2^{(i-2)} \geq \text{size}$ :
  - 重复以下操作，直到  $i > 1$ :
    - 将  $\text{freelists}[\text{LAYER\_NUM} - i].\text{state} \leftarrow \text{SPLIT}$
    - $i \leftarrow i - 1$
    - 更新左右子节点状态:
      - $\text{right\_ch} \leftarrow \text{nodes}[\text{RIGHT}(\text{id})]$
      - $\text{right\_ch.state} \leftarrow \text{INLIST}$

- $\text{left\_ch} \leftarrow \text{nodes}[\text{LEFT}(\text{id})]$
- $\text{left\_ch.state} \leftarrow \text{INLIST}$

### 3. 更新最终分配的节点状态:

- 将  $\text{freelists}[\text{LAYER\_NUM} - i].\text{state} \leftarrow \text{ALLO}$

## 结束

- 返回分配的内存地址或 0（如果失败）。

## 4 YuanShen OS的回收算法

回收算法是YuanShen OS最大的亮点。我们使用刚才那个很简单的例子说明原理，不过稍作修改，我们假设刚才的request size其实是2,如图所示：

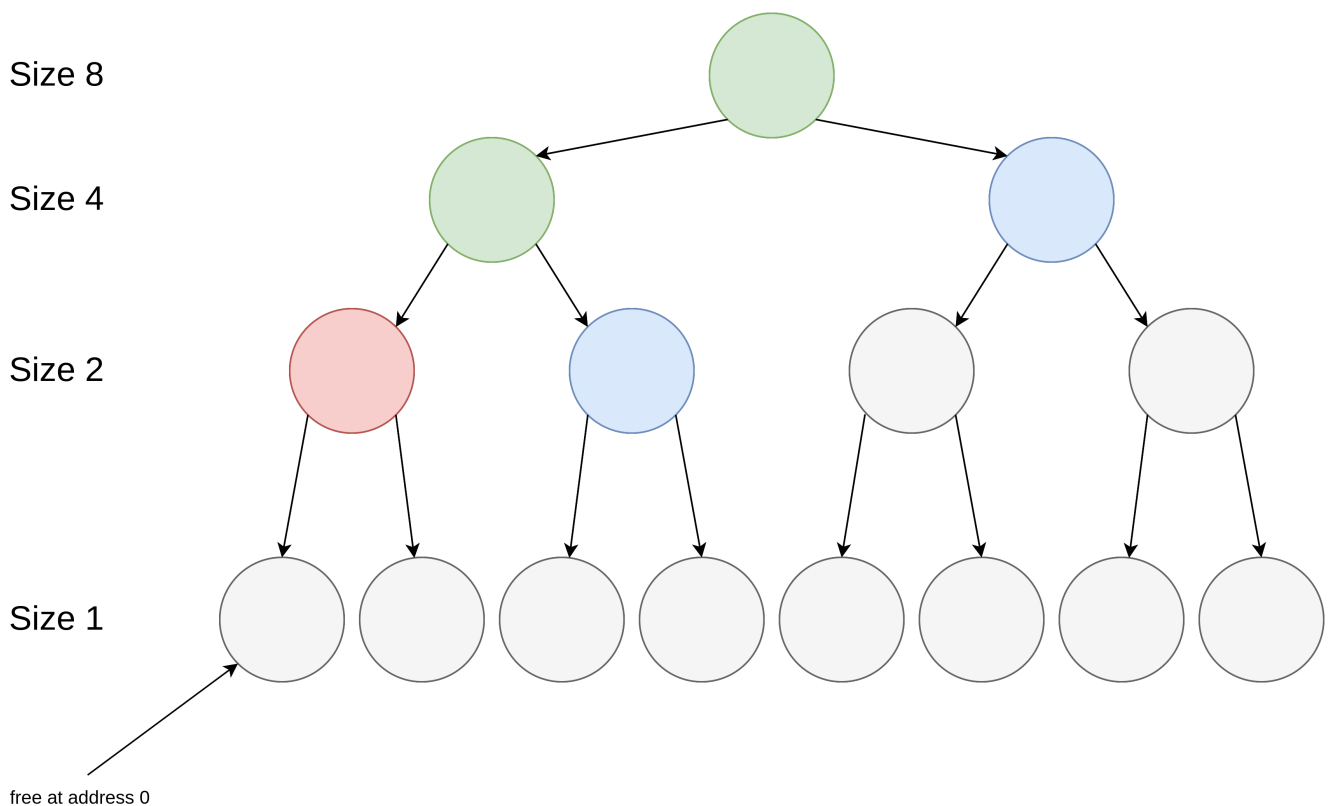


Fig6. 回收一个长为2的区间(原创)

### 4.1 查找被分配出去的结点

假设我们要归还刚才分配出去的结点，但是我们只有它对应的区间的首地址，在这里也就是伙伴系统的基地址，对应0的偏移量，怎样才能找到它对应的结点呢？

方法是分配出去的每一个地址都对应二叉树最下面一层的结点，例如0地址对应最左的那个结点。既然我们要归还的地址为0,我们就可以从那个结点出发，沿着到根结点的路径自下而上，直到遇到一个结点的状态为ALLO(红色)，这个结点就必然对应我们分配出去的区间，于是我们把它标记成INLIST(蓝色)。

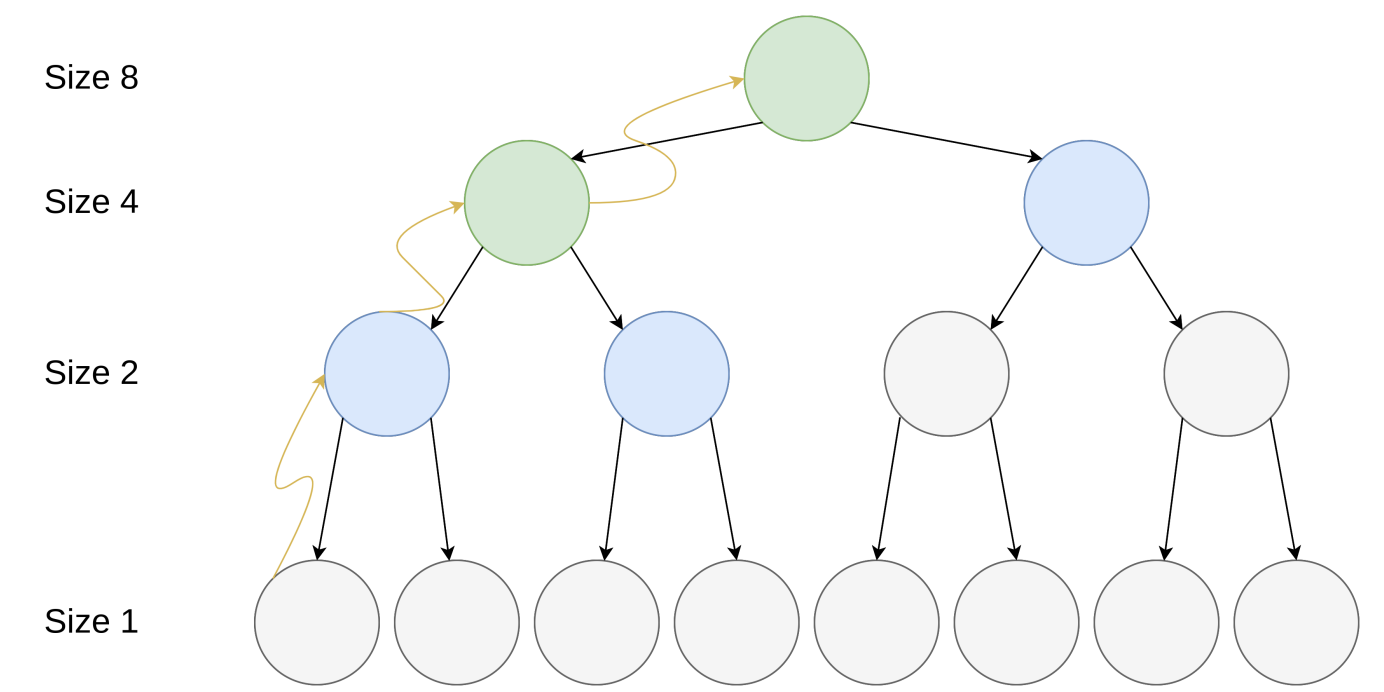


Fig7. 自下而上合并区间(原创)

这时根据二叉树的组织结构，我们可以很轻松的判断当前结点对应的兄弟结点的状态，如果它也是INLIST，那么我们就可以合并这一对伙伴区间

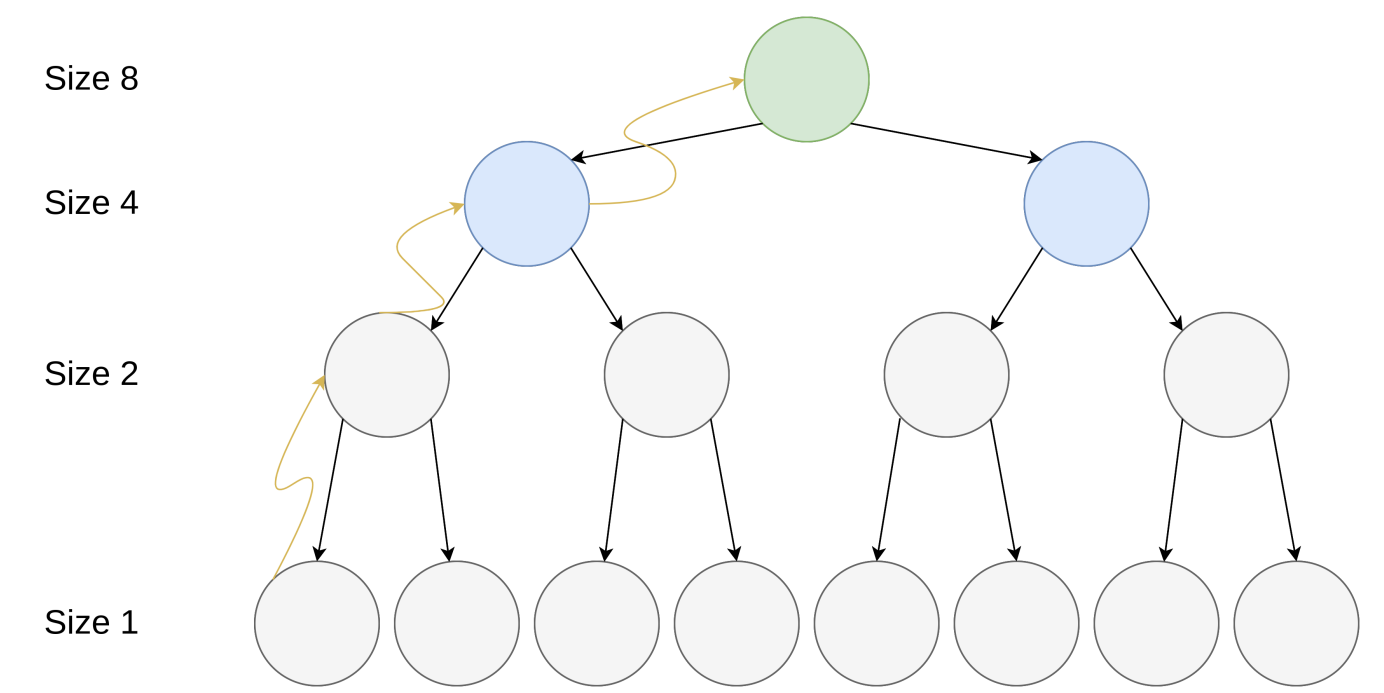


Fig8. 继续合并(原创)

最后我们得到下图，黄线展示了我们自底向上的查询与合并路径，如图所示:

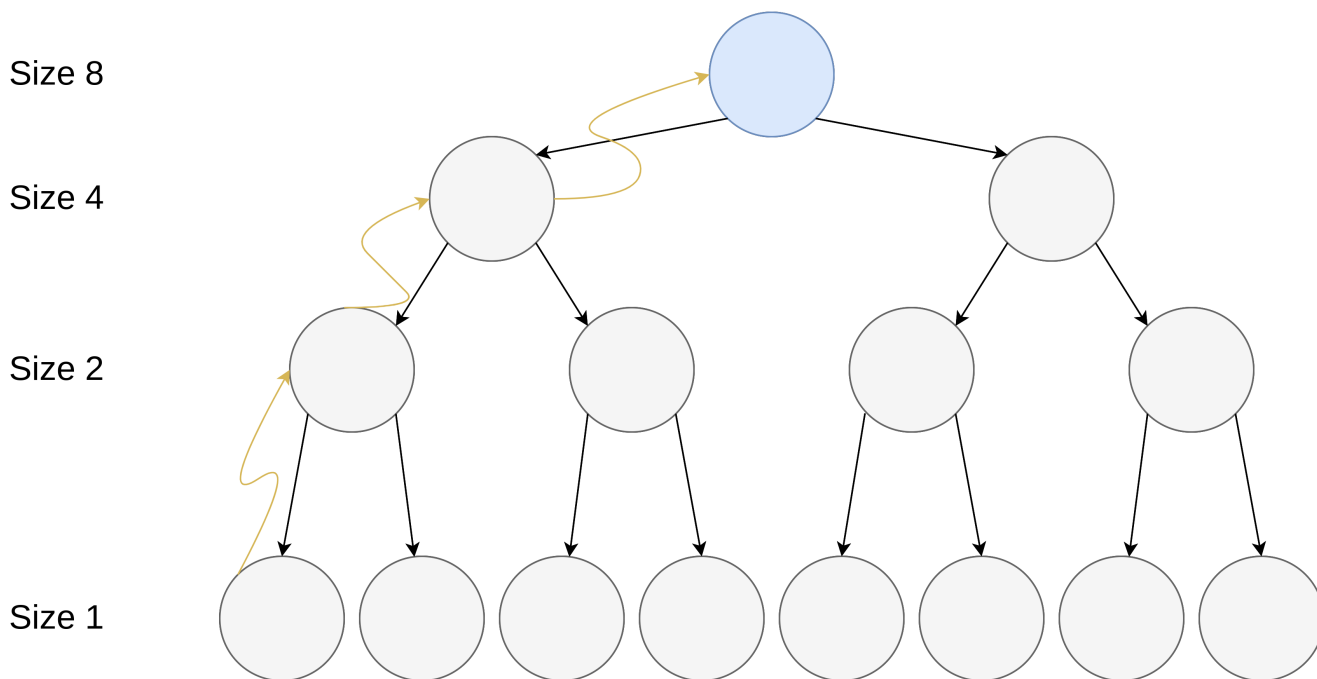


Fig9. 最终的合并结果(原创)

可以看到所有的小区重新被合并为最大的那个区间。

对应的伪代码如下所示：

函数：buddy\_free\_

#### 输入

- addr: 要释放的内存地址。

#### 步骤

##### 1. 计算偏移和索引:

- $\text{uint32 offset} \leftarrow (\text{addr} - \text{bdsys.base\_addr}) \gg 12$
- $\text{uint32 idx} \leftarrow (1 \ll (\text{LAYER\_NUM} - 1)) - 1 + \text{offset}$
- $\text{uint8 i} \leftarrow 1$

##### 2. 找到分配状态的节点:

- 重复以下操作，直到找到状态为 ALLO 的节点:
  - 如果  $\text{idx} == 0$ :
    - 调用 `panic("buddy_free_: can't return")`
  - $\text{idx} \leftarrow \text{PARENT}(\text{idx})$
  - $i++$

##### 3. 释放节点:

- $\text{bdsys.nodes}[\text{idx}].\text{state} \leftarrow \text{INLIST}$  // 将节点状态设置为 INLIST

##### 4. 合并节点:



- uint32 parent\_id
- int flag  $\leftarrow$  0
- 重复以下操作，直到 flag == 1 或 idx == 0:
  - parent\_id  $\leftarrow$  PARENT(idx)
  - struct tree\_node \*right\_ch  $\leftarrow$  &bdsys.nodes[RIGHT(parent\_id)]
  - struct tree\_node \*left\_ch  $\leftarrow$  &bdsys.nodes[LEFT(parent\_id)]
  - 如果 left\_ch->state == right\_ch->state 且 right\_ch->state == INLIST:
    - left\_ch->state  $\leftarrow$  UNUSED // 将左子节点状态设置为 UNUSED
    - right\_ch->state  $\leftarrow$  UNUSED // 将右子节点状态设置为 UNUSED
    - bdsys.nodes[parent\_id].state  $\leftarrow$  INLIST // 将父节点状态设置为 INLIST
  - i++
  - idx  $\leftarrow$  parent\_id

## 5 测试

特别注意：由于后面实现SLUB分配器又对内存分配的逻辑进行了大幅度修改，在最新的分支将不再使用这个测试，而使用更为严格的kmmalloc测试。本测试是在分支feature/buddy\_system下进行的，请不要在新的分支里使用本测试。

### TEST\_FULL\_BUDDY\_SYSTEM测试

在main.c内部启用宏，会进行多项测试，下面我们只列举最强的那一条。

```
#define TEST_FULL_BUDDY_SYSTEM
#define TEST_BUDDY_SYSTEM
```

测试程序的运行结果如下图所示:

```
Free lists:
Level 0: [ID: 0]
Level 1: (empty)
Level 2: (empty)
Level 3: (empty)
Level 4: (empty)
Level 5: (empty)
Level 6: (empty)
Level 7: (empty)
Level 8: (empty)
Level 9: (empty)
Level 10: (empty)
Stress testing allocations and frees...
Allocated block of size 8 at address: 0x1000
Allocated block of size 8 at address: 0x9000
Allocated block of size 8 at address: 0x11000
Allocated block of size 8 at address: 0x19000
Allocated block of size 8 at address: 0x21000
Allocated block of size 8 at address: 0x29000
Allocated block of size 8 at address: 0x31000
Allocated block of size 8 at address: 0x39000
Allocated block of size 8 at address: 0x41000
Allocated block of size 8 at address: 0x49000
Allocated block of size 16 at address: 0x51000
Allocated block of size 16 at address: 0x61000
Allocated block of size 16 at address: 0x71000
Allocated block of size 16 at address: 0x81000
Allocated block of size 16 at address: 0x91000
Allocated block of size 16 at address: 0xa1000
Allocated block of size 16 at address: 0xb1000
Allocated block of size 16 at address: 0xc1000
Allocated block of size 16 at address: 0xd1000
Allocated block of size 16 at address: 0xe1000
Allocated block of size 32 at address: 0x101000
Allocated block of size 32 at address: 0x121000
Allocated block of size 32 at address: 0x141000
Allocated block of size 32 at address: 0x161000
Allocated block of size 32 at address: 0x181000
Allocated block of size 32 at address: 0x1a1000
Allocated block of size 32 at address: 0x1c1000
Allocated block of size 32 at address: 0x1e1000
Allocated block of size 32 at address: 0x201000
Allocated block of size 32 at address: 0x221000
Allocated block of size 64 at address: 0x241000
Allocated block of size 64 at address: 0x281000
Allocated block of size 64 at address: 0x2c1000
Allocated block of size 64 at address: 0x301000
Allocated block of size 64 at address: 0x341000
Allocated block of size 64 at address: 0x381000
Allocated block of size 64 at address: 0x3c1000
Free lists:
Level 0: (empty)
Level 1: (empty)
```

Fig10. TEST\_FULL\_BUDDY\_SYSTEM 测试运行结果

一开始level0的free list有一个结点，表示整个区间。之后测试进行了大量的分配测试，最后集中进行统一回收，结果如下图所示：

```

Allocated block of size 32 at address: 0x121000
Allocated block of size 32 at address: 0x141000
Allocated block of size 32 at address: 0x161000
Allocated block of size 32 at address: 0x181000
Allocated block of size 32 at address: 0x1a1000
Allocated block of size 32 at address: 0x1c1000
Allocated block of size 32 at address: 0x1e1000
Allocated block of size 32 at address: 0x201000
Allocated block of size 32 at address: 0x221000
Allocated block of size 64 at address: 0x241000
Allocated block of size 64 at address: 0x281000
Allocated block of size 64 at address: 0x2c1000
Allocated block of size 64 at address: 0x301000
Allocated block of size 64 at address: 0x341000
Allocated block of size 64 at address: 0x381000
Allocated block of size 64 at address: 0x3c1000
Free lists:
Level 0: (empty)
Level 1: (empty)
Level 2: (empty)
Level 3: (empty)
Level 4: (empty)
Level 5: (empty)
Level 6: [ID: 78]
Level 7: (empty)
Level 8: (empty)
Level 9: (empty)
Level 10: (empty)
Freeing all blocks in reverse order...
Freeing block at address: 0x3c1000
Freeing block at address: 0x381000
Freeing block at address: 0x341000
Freeing block at address: 0x301000
Freeing block at address: 0x2c1000
Freeing block at address: 0x281000
Freeing block at address: 0x241000
Freeing block at address: 0x221000
Freeing block at address: 0x201000
Freeing block at address: 0x1e1000
Freeing block at address: 0x1c1000
Freeing block at address: 0x1a1000
Freeing block at address: 0x181000
Freeing block at address: 0x161000
Freeing block at address: 0x141000
Freeing block at address: 0x121000
Freeing block at address: 0x101000
Freeing block at address: 0xe1000
Freeing block at address: 0xd1000
Freeing block at address: 0xc1000
Freeing block at address: 0xb1000
Freeing block at address: 0xa1000
Freeing block at address: 0x91000
Freeing block at address: 0x81000
Freeing block at address: 0x71000
Freeing block at address: 0x61000
Freeing block at address: 0x51000
Freeing block at address: 0x49000
Freeing block at address: 0x41000
Freeing block at address: 0x39000
Freeing block at address: 0x31000
Freeing block at address: 0x29000
Freeing block at address: 0x21000
Freeing block at address: 0x19000
Freeing block at address: 0x11000
Freeing block at address: 0x9000
Freeing block at address: 0x1000
Free lists:
Level 0: [ID: 0]
Level 1: (empty)
Level 2: (empty)
Level 3: (empty)
Level 4: (empty)
Level 5: (empty)
Level 6: (empty)
Level 7: (empty)
Level 8: (empty)
Level 9: (empty)
Level 10: (empty)
Intensive test for buddy_free_ completed.

```

Fig11. 测试中最终回收完毕的结果

最后的结果是free list又回到最开始的状态，说明所有分配的页都被正确的回收了。

测试代码节选如下所示，主要内容就是进行大量的分配最后回收，观察每一步区间的状态是否符合理论：

```

void test_buddy_free() {
    printf("Testing buddy_free_ intensively...\n");

    buddy_init_(0x1000); // Initialize the buddy system

    // Allocate various block sizes
    unsigned long long addr1 = buddy_alloc_(16);
    unsigned long long addr2 = buddy_alloc_(32);
    unsigned long long addr3 = buddy_alloc_(64);
    unsigned long long addr4 = buddy_alloc_(8);

```

```
    if (addr1 != (unsigned long long)-1) printf("Allocated block at
address: 0x%llx\n", addr1);
    if (addr2 != (unsigned long long)-1) printf("Allocated block at
address: 0x%llx\n", addr2);
    if (addr3 != (unsigned long long)-1) printf("Allocated block at
address: 0x%llx\n", addr3);
    if (addr4 != (unsigned long long)-1) printf("Allocated block at
address: 0x%llx\n", addr4);

    print_free_lists();

    // Free some blocks and check the free lists
    printf("Freeing block at address: 0x%llx\n", addr1);
    buddy_free_(addr1);
    print_free_lists();

    printf("Freeing block at address: 0x%llx\n", addr3);
    buddy_free_(addr3);
    print_free_lists();

    // Allocate again to observe fragmentation handling
    unsigned long long addr5 = buddy_alloc_(16);
    printf("Allocated block at address: 0x%llx (expected to reuse a split
block)\n", addr5);
    print_free_lists();

    // Free all remaining blocks to test coalescing
    printf("Freeing block at address: 0x%llx\n", addr2);
    buddy_free_(addr2);
    print_free_lists();

    printf("Freeing block at address: 0x%llx\n", addr4);
    buddy_free_(addr4);
    print_free_lists();

    printf("Freeing block at address: 0x%llx\n", addr5);
    buddy_free_(addr5);
    print_free_lists();

    // Stress test with maximum allocations and coalescing
    printf("Stress testing allocations and frees...\n");
    unsigned long long addresses[100];
    int allocation_count = 0;

    for (int size = 8; size <= 64; size *= 2) {
        for (int i = 0; i < 10; i++) {
            unsigned long long addr = buddy_alloc_(size);
            if (addr != (unsigned long long)-1) {
                addresses[allocation_count++] = addr;
                printf("Allocated block of size %d at address: 0x%llx\n",
size, addr);
            }
        }
    }
}
```

```
print_free_lists();

printf("Freeing all blocks in reverse order...\n");
for (int i = allocation_count - 1; i >= 0; i--) {
    printf("Freeing block at address: 0x%llx\n", addresses[i]);
    buddy_free_(addresses[i]);
}

print_free_lists();

printf("Intensive test for buddy_free_ completed.\n");
}
```

## 参考资料

[1] "Buddy memory allocation." Wikipedia. [https://en.wikipedia.org/wiki/Buddy\\_memory\\_allocation](https://en.wikipedia.org/wiki/Buddy_memory_allocation).