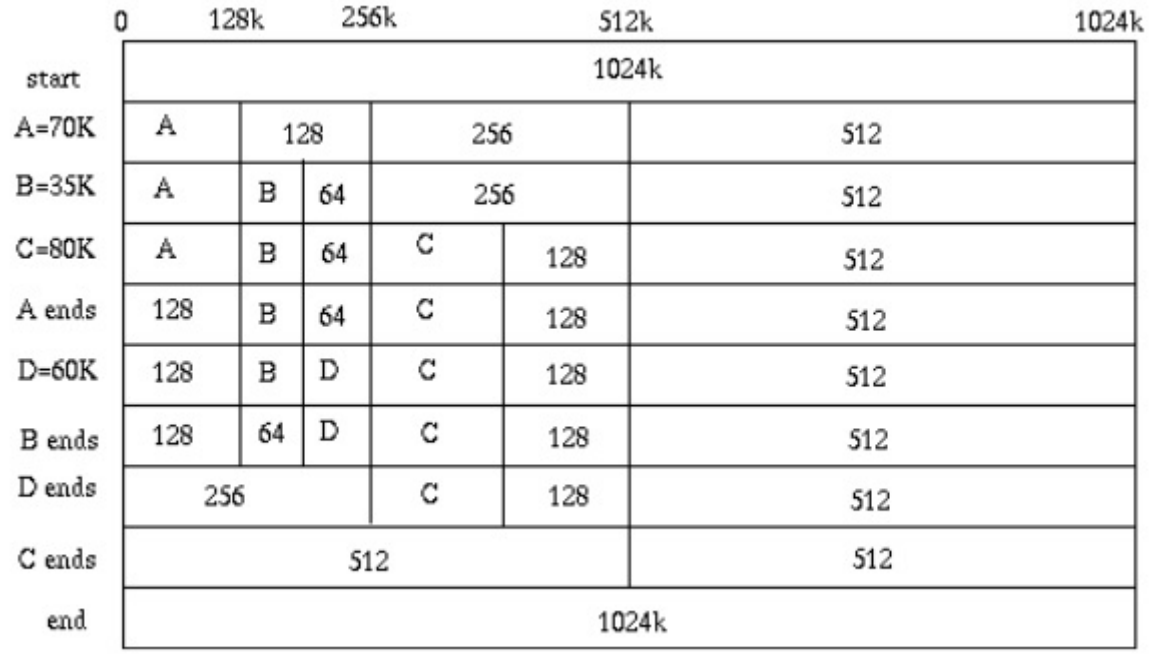
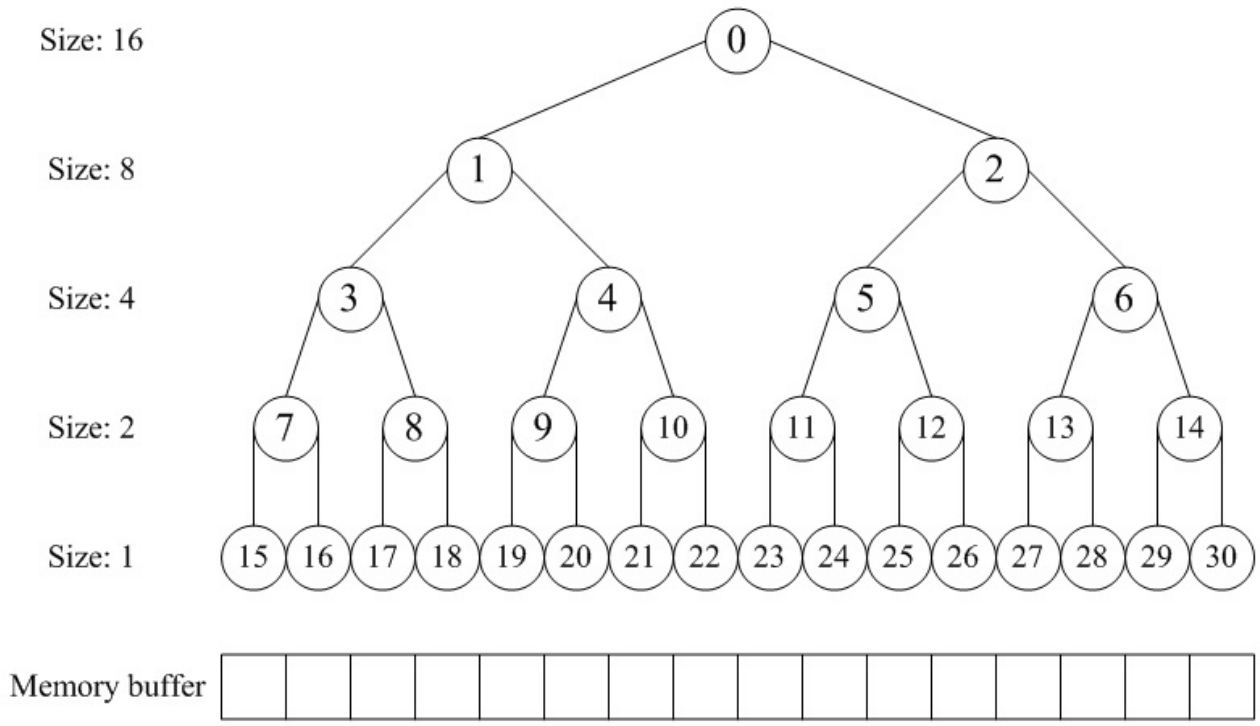
Buddy System 设计文档

本次实验的challenge1部分，我们实现了Buddy System。Buddy System是linux内核中的内存管理技术，用于管理系统中的物理内存。它是一种空间分配算法，用于分配内存块并解决内存碎片化的问题。Buddy System的主要想法是将内存按2的幂进行划分，根据进程的需求给出同最佳匹配的大小。其优点是快速搜索合并（O(logN)时间复杂度）以及低外部碎片（最佳适配best-fit）；其缺点是内部碎片和可能导致利用率较低，因为按2的幂划分块，如果碰上65单位大小，那么必须划分128单位大小的块。



本次实现的Buddy System会以页为最小单位，进行内存的划分。

利用一个类似二叉树的层级结构可以方便地实现一个Buddy System内存管理系统，可以做到减少内存碎片，并提供高效的内存分配和释放机制。



我们采用数组的方式来实现内存管理的二叉树，其不同层的结点会代表不同大小的内存块，总的结点数会有（2*页数-1），在分配时自顶向下找到第一块大小匹配的结点即可，而在释放时则自底向上，实现内存块的合并。

1. 管理器实现

在整个buddy system中，首先定义了一个伙伴管理器：

```
struct Buddy
{
    unsigned *size;
    struct Page *mem_tree;
    unsigned total_size;

} buddy_manager;
```

其中size是二叉树结点数组的起始地址，可以通过`buddy_manager.size[i]`来访问第i个结点的信息；而`mem_tree`则是内存中页面排布的起始地址，通过此基地址来管理页面；`total_size`是整个二叉树所管理的内存的页面数 具体的内存管理的实现办法如下：

2. 初始化内存池：

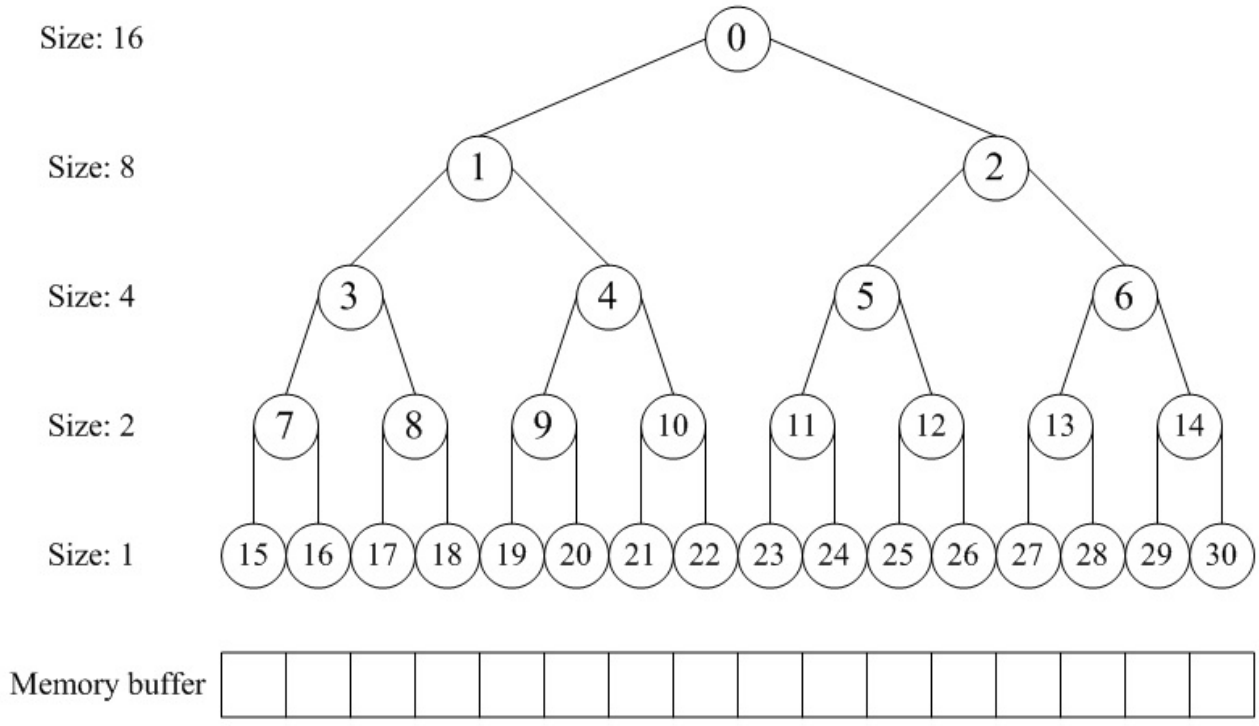
- 在Buddy System中，会要求整块可用内存大小是2的幂，因此在初始化的buddy_init_memmap中，会要求参数n为2的幂，如果不是，则会自动调整为最佳匹配n的2的幂次大小；此外，还会将内存管理的二叉树进行初始化；

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

    struct Page *p = base;
    for (; p != base + n; p++)
    {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);

    }
    p = base + n; //p指向page的最末尾地址
    buddy_manager.mem_tree = base;

    //紧接着n个page，我们将记录结点的数组从此处排开
    buddy_manager.size = (unsigned *)p;
    base->property = n; // 从base开始有n个可用页
    buddy_manager.total_size = round_up_n;
```

紧接着，我们对管理内存页面的二叉树进行初始化，也就是对`buddy_manager.size`进行初始化：

```
unsigned node_size = 2 * round_up_n;
for (int i = 0; i <2 * round_up_n - 1; ++i)
  {
      if (IS_POWER_OF_2(i + 1))
      {
          node_size /= 2;
      }
      buddy_manager.size[i] = node_size;

  }
```

这是一个对二叉树自顶向下的初始化，最顶部的结点有round_up_n个页面可用，每一层结点代表的可用空间是上一层的1/2，而当IS_POWER_OF_2(i + 1)为真时，说明进入了下一层结点，node_size就会除以2。通过这样的循环，可以初始化出一个如图的二叉树：



3. 内存分配：

- 在申请n块内存时，会首先用IS_POWER_OF_2检查n是否是2的幂次，不是则分配一块最佳匹配的2的幂次大小的内存块
- 其次会进行判断，当前是否能分配出合适大小的内存：

```
if (n > nr_free) //大于剩余空间之和
    return NULL;

if (buddy_manager.size[index] < n)
//此时index = 0，buddy_manager.size[0]代表能分配出的
//最大的连续内存空间
    return NULL;
```

- 紧接着，通过一个自顶向下的搜索，找到一个best_fit的内存块：

```
for (node_size = buddy_manager.total_size; node_size != n; node_size /= 2)
    {
        if (buddy_manager.size[LEFT_LEAF(index)] >= n)
            index = LEFT_LEAF(index);
        else
            index = RIGHT_LEAF(index);
    }
```

首先将node_size赋值为整块内存空间大小，代表最顶层的根结点，每迭代一次，node_size会除以2，代表进入下一层，直到内存块大小和n相同的一层，找到对应的结点；

- 找到了这块结点后，需要将该结点代表的可用内存空间标记为0，并且根据结点的序号，计算出对应的内存块索引，即：

```
 buddy_manager.size[index] = 0; //置0后代表该结点的可用空间为0
 offset = (index + 1) * node_size - buddy_manager.total_size;
 //相当于一个映射运算，通过结点序号index计算出内存块索引offset
```

- 在找到要分配出的内存块起始地址后，还需要关键的一步：

```
while (index)
{
  index = PARENT(index);
  buddy_manager.size[index] = MAX(buddy_manager.size[LEFT_LEAF(index)],
buddy_manager.size[RIGHT_LEAF(index)]);

}
```

从结点index开始，向上调整其每个祖父结点的可用空间，每个结点取其子结点的最大值，代表可以分配的最大空间。

- 最后，将分配出的n个页全部标记为reserved，并且将分配的内存起始地址`buddy_manager.memtree + offset`返回

```
    struct Page *base = buddy_manager.mem_tree + offset;
    struct Page *page;

    // 将每一个取出的块由空闲态改为保留态
    for (page = base; page != base + n; page++)
    {
        ClearPageProperty(page);
    }
    nr_free -= n;
```

```
    cprintf("alloc done at %u with %u pages\n",offset,n);
    return base;
```

4.内存释放：

- 第一步与分配类似，检查释放空间n的值是否合法，不是2的幂的进行fix_size
- 与分配页面时从结点序号计算内存块索引相反，释放内存时，需要通过内存块索引，计算出对应的二叉树的结点序号：

```
int offset = (base - buddy_manager.mem_tree);
index = buddy_manager.total_size + offset - 1;
//此时index是二叉树的一个最底层结点序号
node_size = 1;
 while (node_size != n)//自底向上，找到要free的结点的序号
{
    node_size *= 2;
    index = PARENT(index);
    if (index == 0)
        return;
}
buddy_manager.size[index] = node_size;
//将该结点的可用空间标记为n，即node_size
```

直到这一步，完成了对这一块大小为n的内存的空间释放

- 下一步则是完成内存空间的合并整理，递归地将相邻且大小相同的内存空间合并：

```
while (index) //自index向上的合并
{
    index = PARENT(index);
    node_size *= 2;
    left_longest = buddy_manager.size[LEFT_LEAF(index)];
    right_longest = buddy_manager.size[RIGHT_LEAF(index)];
    if (left_longest + right_longest == node_size){
    //相邻且大小相同，合并
        buddy_manager.size[index] = node_size;
    }
    else //调整为最大可分配空间
        buddy_manager.size[index] = MAX(left_longest, right_longest);
}
```

至此，释放操作结束。以上的alloc和free的时间复杂度都是`O(logN)`，保证了程序运行性能，极简地实现了通过一颗二叉树管理内存空间。

5. Check 程序验证

在check程序中，我们首先声明了五个page类型的指针，全部初始化为空，用于后续的验证。

```
    struct Page *p0, *p1, *p2, *p3, *p4;
    p0 = p1 = p2 = p3 = p4 = NULL;
```

变量初始化后，我们开始进行验证。

- 首先进行alloc_page的单页分配检测

将p0,p1,p2分别指向分配单页后的内存起始地址，按照我们的预期p0,p1,p2此时应该不同并且它们被reference的次数应该都是0。

并且p0，p1，p2理应按顺序排布，所以应该依次为下一个结构体的前一个元素。我们调用page2pa函数将其page结构体转成了page所在的真正的物理地址，按照我们的想法这里不应该超过物理地址的上限。(npage定义见注释)

```
    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(p1==p0+1 && p2 == p1+1);  //页面地址关系

    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);

    // in pmm.c
    /*
    uint64_t maxpa = mem_end;

    if (maxpa > KERNTOP) {
        maxpa = KERNTOP;
    }

    npage = maxpa / PGSIZE;
    */
```

- 指定大小分配检测

我们首先释放p0，p1，p2。再次分配p1，p2，p3为大小512，512，1024的内存空间的起始地址，并检查相邻关系。理论上，应从第0个页开始分配页面，p1、p2、p3之间都相距512的大小，那么应该满足我们的断言：

```
    free_page(p0);
    free_page(p1);
    free_page(p2);

    p1 = alloc_pages(512); //p1应该指向最开始的512个页
    p2 = alloc_pages(512);
```

```
    p3 = alloc_pages(1024);
    /*
    ---------------------------------
    |     512       |       512      |
    ---------------------------------
    p1              p2                p3
    /*

    assert(p3 - p2 == p2 - p1);//检查相邻关系
```

接着我们按照交叉的方式释放掉内存，并通过后续的分配和断言验证正确性。

```
    free_pages(p1, 256);
    free_pages(p2, 512);
    free_pages(p1 + 256, 256);
    free_pages(p3,1024);
    //检验释放页时，相邻内存的合并
```

将p1指向的512的页分两次释放，可以检测释放过程中内存是否正确合并。

我们知道p1 = 0(页面起始地址)，而且释放过程中并没有修改其值，而在释放所有页面后，内存应该从0开始全部可用并且合并为整块。 那么第一次分配内存给p0后，p0应该也是0，与分配大小无关。

```
    p0 = alloc_pages(8192);

    assert(p0 == p1); //重新分配，p0也指向最开始的页
```

此时为p1、p2分配内存，此时p1应该是8192、而p2应该是（8192+128）、p3应该是（8192+128+128）

```
    p1 = alloc_pages(128);
    p2 = alloc_pages(64);

    assert(p1 + 128 == p2);// 检查是否相邻

    p3 = alloc_pages(128);
    /*
    ---------------------------------------------------
    |       128           |     64     |     64     |
    ---------------------------------------------------
    p1                     p2                        p3


    /*

    //检查p3和p1是否重叠
    assert(p1 + 256 == p3);
```

释放p1,又为p4分配空间，应该分配刚才p1占用的空间，此时p4应该是8192

```
    //释放p1
    free_pages(p1, 128);

    p4 = alloc_pages(64);
    assert(p4 + 128 == p2);
    // 检查p4是否能够使用p1刚刚释放的内存
```

再释放分配p3，因为p4没有用完p1的128，刚好剩下64给p3, 所以p3应该紧接着p4，是（8192+64）

```
    free_pages(p3, 128);
    p3 = alloc_pages(64);

    // 检查p3是否在p2、p4之间
    assert(p3 == p4 + 64 && p3 == p2 - 64);
```

数值关系全部成立，assert测试通过。