Lab2

操作系统实验二

学号：2313501 姓名：杨楠欣 ;学号：2213230 姓名：向宇涵； 学号：2311366 姓名：邵莫涵

练习1：理解 first-fit 连续物理内存分配算法

1. 算法概述

First-Fit 连续物理内存分配算法是操作系统中管理连续物理内存的经典动态分区分配策略，核心逻辑是通过维护一个仅空闲块起始页加入、按物理地址从低到高有序排列的双向空闲链表，实现对空闲内存的高效分配与回收。其中，仅空闲块的起始页会记录块内总页数并标记PG_property，块内其他页不存储块信息。

当收到内存分配请求时，算法从空闲链表表头开始遍历，找到第一个块内总页数大于等于请求页数的空闲块。若块大小等于请求页数，则直接将该块从链表删除。若块大于请求页数，则拆分出 “分配块” 和 “剩余空闲块”（插回原链表位置）。

释放内存时，会将释放的连续页按地址有序插入空闲链表，并自动合并与释放块物理地址相邻的前后空闲块以减少外部内存碎片。其优势在于实现简洁、分配效率高，即无需遍历全部空闲块，虽存在外部碎片问题，但可通过块合并机制部分缓解，是平衡性能与实现复杂度的常用内存分配方案。

2. 函数分析

2.1 default_init

功能：初始化空闲链表和空闲页数。

代码块

```
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

要理解该函数的作用，需从物理内存管理的基础数据结构说起，这些结构共同构成了空闲内存块的管理框架。

- 双向链表节点（定义于 list.h）

代码块

```
struct list_entry {
    struct list_entry *prev, *next;
};
typedef struct list_entry list_entry_t; //起别名
```

在物理内存管理中，每个空闲块通过该结构的节点（嵌入在 struct Page 中）链接在一起，形成空闲块链表。

> 一个**空闲块**是由多个物理地址连续的物理页组成的，不同空闲块之间需要用链表管理，而链接的载体是每个空闲块的**起始页**。在 struct Page 中，会嵌入一个 list_entry_t 类型的成员 page_link。对于一个空闲块，只有它的起始页的 page_link 会被加入到 free_list（空闲块链表）中。
>
> - 每个物理页的描述符（定义于 memlayout.h）
>
> 代码块
>
> ```
> struct Page {
>     int ref;
>     uint64_t flags;
>     unsigned int property;
>     list_entry_t page_link;   // free list link
> };
> ```

- 链表初始化函数（定义于 list.h）

代码块

```
static inline void list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;
}
```

该函数用于初始化一个链表节点，将其前驱和后继指针都指向自身，表示链表中暂时没有其他节点。

- 空闲内存管理结构（定义于 memlayout.h）

代码块

```
typedef struct {
    list_entry_t free_list;     // 空闲块链表的表头
    unsigned int nr_free;       // 空闲物理页的总数量
} free_area_t;
```

free_list 作为空闲块链表的表头，类型为 list_entry_t，所有空闲块通过各自的 list_entry_t 节点挂载到该链表上，实现对空闲块的统一管理。

nr_free 用于记录系统中所有空闲物理页的总数，用于快速判断是否有足够内存满足分配请求。

- 全局空闲内存管理器实例

代码块

```
static free_area_t free_area;
#define free_list (free_area.free_list)  // 简化对 free_area 中 free_list 的访问
```

现在，我们就可以很好地理解 default_init 函数的作用了。

通过 list_init(&free_list) 初始化 free_area 中的链表表头 free_list，使其成为一个自循环的空链表，因为此时尚无任何空闲块。

将空闲页总数初始化为 0，因为系统启动初期尚未登记空闲物理内存。

2.2 default_init_memmap

功能：将从起始物理页base开始的n个连续物理页初始化为空闲块（重置各页属性、在起始页记录块大小并标记为空闲块起始页、更新空闲页总数），并按物理地址从低到高的顺序插入空闲链表。

传入的参数：

- 要初始化的空闲块的起始物理页描述符的指针
- 要初始化的空闲块包含的连续物理页的总数量

代码块

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);                    // 页数至少为1
    struct Page *p = base;            // 初始化遍历指针p
    /* 遍历这段物理页，初始化每个页的基础属性 */
    for (; p != base + n; p ++) {
        // 确保该页初始为“保留态”，避免重复初始化已用页，或操作不属于该空闲块的页。
        assert(PageReserved(p));
        p->flags = p->property = 0; //重置页的标志位和property（清除保留位，先都不设成起始页）
        set_page_ref(p, 0);         // 重置页的引用数，空闲页无任何进程引用，设为0
    }
    base->property = n;     // 在空闲块的【起始页】的property字段，记录整个空闲块的总页数n
    SetPageProperty(base); // 设置PG_property标志位，表明该页是某个空闲块的【起始页】
    nr_free += n;           // 更新系统空闲页总数，将当前空闲块的 n页加入总计数

    /* 将初始化好的空闲块，按【物理地址从低到高】的顺序插入链表 */
    if (list_empty(&free_list)) {  // 首次初始化时，空闲链表为空
        list_add(&free_list, &(base->page_link));// 将空闲块的起始页节点（page_link）加入链表
```

```
    } else {                        // 链表非空时，遍历链表找合适的插入位置，要保持地址有序
        list_entry_t* le = &free_list;// 初始化链表遍历指针le
        while ((le = list_next(le)) != &free_list) {// 遍历链表所有节点，直到回到表头
            // 将链表节点（list_entry_t）转换为对应的Page指针
            struct Page* page = le2page(le, page_link);
            /* 比较地址：当前要插入的空闲块起始页（base），是否比已有块起始页（page）小 */
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
            // 遍历到最后一个节点（要插入的地址是目前最大），将其插入到链表末尾
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

为了更好地理解以上函数，对用到的一些工具函数予以介绍。

- 通用结构体定位宏（定义于 defs.h）

代码块

```
#define to_struct(ptr, type, member)                                \
    /* 用成员指针 ptr 减去成员在 type 结构体中的偏移量，得到整个结构体的起始地址，
    再强转为 type* 类型。*/
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

通过结构体中某个成员的指针（ptr），计算出整个结构体的指针。

- 针对 Page 结构体的特化宏（定义于 memlayout.h）

代码块

```
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)
```

专门用于通过 list_entry_t 成员指针（le）获取对应的 struct Page 指针。通过 le2page(le, page_link) 就能快速从链表节点定位到它所属的 struct Page，即空闲块的起始页。

- 链表基础操作函数

```
static inline void list_add(list_entry_t *listelm, list_entry_t *elm) {
list_add_after(listelm, elm); // 实际调用后插函数
}
```

在 listelm 【之前】插入 elm 节点

```
static inline void list_add_before(list_entry_t *listelm, list_entry_t *elm) {
__list_add(elm, listelm->prev, listelm); // 内部实现双向链表的指针调整
}
```

2.3 default_alloc_pages

功能：按首次适配算法，从空闲块链表表头开始遍历找到第一个大小不小于n的空闲块，若找到则将其从链表删除，若块大于n页则拆分出剩余空闲块并插回链表，更新空闲页总数并清除分配块起始页的标记后返回起始页指针，若无合适块则返回NULL。

传入的参数：

- 需要分配的连续物理页数量。

代码块

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);     // 页数至少为1
    if (n > nr_free) { // 需要分配的页数超过系统当前空闲页总数，直接返回NULL
        return NULL;
    }
    struct Page *page = NULL; // 声明指针page，用于存储找到的合适空闲块的起始页
    list_entry_t *le = &free_list;// 声明链表遍历指针le，从空闲块链表的表头（free_list）开始遍历
    while ((le = list_next(le)) != &free_list) {// 遍历直到回到表头
        struct Page *p = le2page(le, page_link);// 将当前链表节点转换为对应的struct Page指针
        if (p->property >= n) {// p->property记录了块内总页数，因为p是【起始页】描述符指针
            page = p;// 找到首个满足条件的空闲块，记录其起始页并退出循环
            break;
        }
    }
    /* 若找到合适的空闲块，执行分配与拆分操作*/
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));// 记录当前空闲块在链表中的前一个节点
        list_del(&(page->page_link));// 将当前空闲块从空闲链表中删除
        if (page->property > n) {// 若空闲块大小大于需要分配的页数，需要拆分出剩余空闲块
```

```
21              struct Page *p = page + n; // 计算剩余空闲块的起始页，即当前块起始页向后偏移n页
22              p->property = page->property - n;// 设置剩余块的大小
23              SetPageProperty(p);// 设置PG_property标志位，表明该页是某个空闲块的【起始页】
24              list_add(prev, &(p->page_link));// 将剩余块插入到原空闲块的位置，即前节点之后
25          }
26          nr_free -= n;// 系统空闲页总数减去已分配的n页
27          ClearPageProperty(page);// 清除已分配块起始页的PG_property标志位
28      }
29      return page;// 返回分配到的起始页指针
30  }
```

2.4 default_free_pages

功能：释放从 base 开始的 n 个连续物理页，将其加入空闲链表并合并相邻空闲块以减少碎片。

传入的参数：

- 要释放的连续物理页的起始页描述符指针
- 要释放的物理页数量

代码块

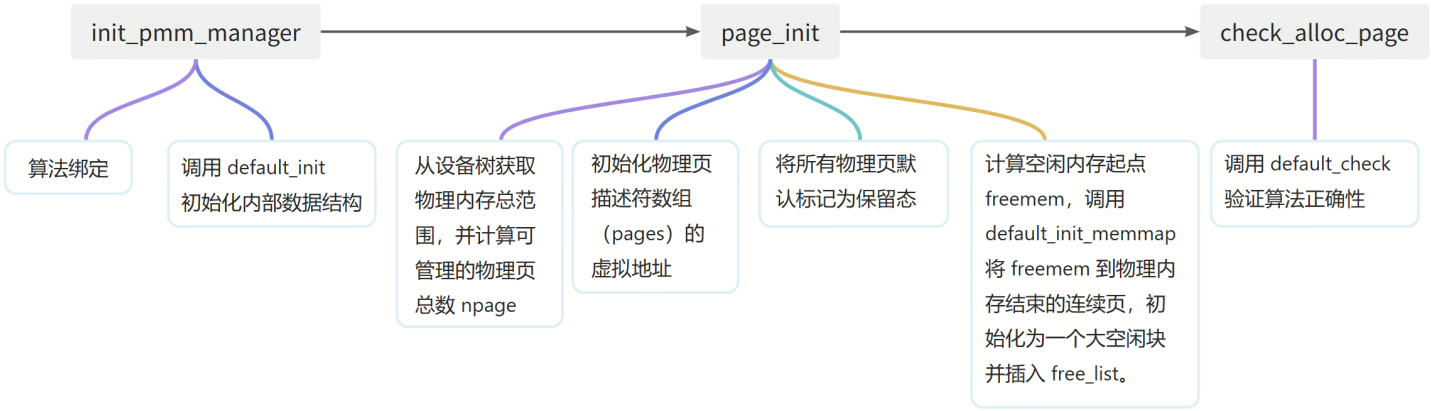
```
1   static void
2   default_free_pages(struct Page *base, size_t n) {
3       assert(n > 0);
4       struct Page *p = base; // 初始化遍历指针p为释放块的起始页地址
5       for (; p != base + n; p ++) {// 遍历要释放的 n个页
6           // 确保当前页是非保留态且非起始页
7           assert(!PageReserved(p) && !PageProperty(p));
8           p->flags = 0;// 重置页的标志位
9           set_page_ref(p, 0);// 重置页的引用数，释放后无进程引用，设为0
10      }
11      base->property = n;// 在释放块的【起始页】的property字段记录总页数n
12      SetPageProperty(base);// 设置PG_property标志位，表明该页是空闲块的起始页
13      nr_free += n; // 更新系统空闲页总数
14
15      /*将释放的空闲块按物理地址从低到高的顺序插入空闲链表*/
16      if (list_empty(&free_list)) {// 链表为空时，直接将释放块的起始页节点（page_link）加入链表
17          list_add(&free_list, &(base->page_link));
18      } else {// 链表非空时，遍历链表找合适的插入位置
19          list_entry_t* le = &free_list;
20          while ((le = list_next(le)) != &free_list) {
21              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) {
                list_add(le, &(base->page_link));
            }
        }
    }

    /*尝试合并释放块与【前一个空闲块】，即低地址相邻块*/
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {// 若前一个节点不是链表表头，即存在前一个空闲块
        p = le2page(le, page_link);// 将前一个节点转换为对应的Page指针
        //前一个块的结束地址（最后一页的下一页）等于释放块的起始地址，即地址连续
        if (p + p->property == base) {
            p->property += base->property;// 合并，更新块大小
            ClearPageProperty(base);// 清除释放块的起始页标志
            list_del(&(base->page_link));// 将释放块从链表中删除
            base = p;// 更新base为合并后的块的起始页
        }
    }

    /*尝试合并释放块（可能已与前块合并，base=p）与【后一个空闲块】，即高地址相邻块*/
    le = list_next(&(base->page_link));
    if (le != &free_list) {// 若后一个节点不是链表表头，即存在后一个空闲块
        p = le2page(le, page_link);
        if (base + base->property == p) {// 同理，检查地址连续
            base->property += p->property;// 更新块大小
            ClearPageProperty(p);// 清除后一个块的起始页标志
            list_del(&(p->page_link));// 将后一个块从链表中删除
        }
    }
}
```

> 注：链表表头（free_list）并非实际的空闲块节点，而是链表的锚点，它不对应任何物理页。

3. 物理内存分配与释放流程

3.1 系统初始化（pmm.c）



3.2 物理内存分配与释放

当用户需要连续物理内存时，调用 alloc_page() 或 alloc_page(n)。

当物理页不再使用时（如进程退出、内核释放缓存），调用 free_page(page) 或 free_pages(base, n)。

4. first fit算法的改进空间

我们知道，First-Fit 算法的核心优势是实现简单、分配时首次匹配即停止，但存在遍历效率随空闲块增多下降、外部碎片较多、易拆分大块导致后续大请求无法满足等问题。

> 外部碎片是指系统中存在大量零散的空闲内存块，但这些块物理地址不连续，导致即使空闲内存总量足够，也无法满足连续物理内存的分配请求的内存浪费现象。

针对以上问题，可以考虑做出如下改进。

4.1 优化遍历效率

按块大小（页数）分类来减少遍历范围。将单一空闲链表拆分为3~4 个大小专属链表，每个链表仅管理特定页数范围的空闲块。分配时按请求页数直接定位到对应链表，无需遍历所有块。释放时根据释放块大小插入对应链表，仍保持各链表内部按物理地址有序，这样不会破坏合并相邻块的基础。

实现仅需在原有 free_area_t 中增加几个链表头。

4.2 减少外部碎片

First-Fit 可能会出现优先拆分更大的块，导致产生更多小碎片的情况，改进方案是：

- 首次遍历先记录首个完全匹配块的地址，若存在的话。
- 若遍历结束找到完全匹配块，则优先分配该块。
- 若无完全匹配块，再按原 First-Fit 逻辑分配首个大于等于请求页数的块。

First-Fit 每次释放都立即合并前后相邻块，若进程频繁申请或释放小块，合并操作的 CPU 开销会累积，改进方案是：

- 为每个空闲块增加待合并计数器，释放时先将块标记为待合并，插入对应大小链表，不立即合并。
- 当待合并块数量达到阈值，或分配大请求时无合适块，再批量遍历链表合并所有相邻的待合并块。

4.3 保护大块资源

将大块链表设置为预留模式，仅当无中小块可用时，才允许拆分大块满足小请求。

拆分大块后，若剩余块大小小于最小保留阈值，则不拆分，直接分配整个大块，避免产生无法再利用的极小碎片。

练习2：实现 Best-Fit 连续物理内存分配算法

因为空闲链表是按照空闲块的地址从低到高连在一起的，所以在用first-fit算法找空闲块时找到的是地址最低的空闲块，而不会考虑这个块的大小，因此我们要设计best-fit算法来找到最适合需求的空闲块，也就是能容纳需求页数最小的空闲块。

在这里，两个算法的链表排序、舒适化与释放以及释放后的合并逻辑是一致的，唯一的差别在于分配内存，first-fit在遇到第一个 `property >= n` 就用，best-fit会扫描全表，在所有 `property >= n` 里挑 **最小** 的那个。

1. alloc_pages

首先是在分配页里面，从链表头开始，逐个查看每个空闲块头p，在遍历之前，先把总的能使用的页数和当作最小候选 `min_size` 。遍历开始，如果查找到能装得下请求的页数的块，并且比当前的最小候选块页数要小，那就更新 `min_size` 为这个块的页数值，把目标 `page` 赋值为这个块的地址；遍历完整张表之后， `page` 就会指向最小可用块。这里如果有多个块的大小相同且都为最小块，那么会取到地址更小的那一块，；如果一个块的大小刚好等于n，后续就直接锁定了。

找到最小可用块之后，还是和first-fit一样把选中的块从链表中摘下。在把剩余页插进空闲链表的时候，从这个块剩余的尾部开始，也就是从 `page+n` 开始，作为新的空闲块，这里依然在原位置附近，保持了地址升序。

best_fit_alloc_pages

```
1   static struct Page *
2   best_fit_alloc_pages(size_t n) {
3       assert(n > 0);
4       if (n > nr_free) {
5           return NULL;
6       }
7       struct Page *page = NULL;
8       list_entry_t *le = &free_list;
9       size_t min_size = nr_free + 1;
10
11      /*LAB2 EXERCISE 2: YOUR CODE*/
12      //遍历空闲链表，找出所有 p->property >= n 中“最小”的那块
13      while ((le = list_next(le)) != &free_list) {
```

```
14          struct Page *p = le2page(le, page_link);
15          assert(PageProperty(p));
16          if (p->property >= n && p->property < min_size) {
17              page = p;
18              min_size = p->property;
19          }
20      }
21
22      if (page != NULL) {
23          // 记录插入位置（旧位置的前驱），删除这个已经选择的块
24          list_entry_t* prev = list_prev(&(page->page_link));
25          list_del(&(page->page_link));
26
27          if (page->property > n) {
28              // 返回前 n 页，保留尾部 remain 为空闲块并回插到原位置
29              struct Page *p = page + n;
30              p->property = page->property - n;
31              SetPageProperty(p);
32              // 挂在原位置的前驱之后即可
33              list_add(prev, &(p->page_link));
34          }
35          nr_free -= n;
36          ClearPageProperty(page);
37      }
38      return page;
39  }
```

2. free_pages

这里与first-fit一致，在释放内存的时候，首先需要逐页清理，对从 `base` 地址开始到 `base+n` 中间的每一页如果不是保留页也不是空闲块头页那就把每一页的 `property(这个块的页数)` 置0。

接着需要把 `base` 设为新的空闲块块头，记录块的大小 `n` 并置位 `PG_property` ,全局空闲页数需要增加 `n` 。到这一步，已经将这个页恢复为空闲了，接下来要进行链表的插入和合并。

接着就需要把这个清零的块插入到空闲链表中，如果是空表，就直接插入；否则就从头扫描，找到**第一个起始地址比 `base` 大**的块，在它前面插入；如果扫描到尾都没找到更大的地址，就直接插入到尾部。这样的存储结果还是按照地址升序排列。

接着需要把链表中的节点的前驱和后继进行合并，与前驱块合并的时候，先找到前驱块头 `p` ，若 `p` 的末尾正好紧贴 `base` 的起始，说明两块物理连续，扩大左边的块的 `property` ，这里 `base` 就不是块头了，把 `base` 的指针指向 `p` ；接着找到此时 `base` 的后继块头，如果 `base` 的末尾正好挨着 `p` 的起始，说明右边也物理连续，就把右块合并进来，增加 `base` 的页数。

```c
static void
best_fit_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
        p->property = 0;
    }
    /*LAB2 EXERCISE 2: YOUR CODE*/
    // 设置块头属性并计数
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    // 按物理地址有序插入
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            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) {
                list_add(le, &(base->page_link));
                break;
            }
        }
    }

    // 向左合并
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        /*LAB2 EXERCISE 2: YOUR CODE*/
        // 若前块紧邻当前块：合并到前块
        if (p + p->property == base) {
            p->property += base->property;    // 1) 前块大小累加
            ClearPageProperty(base);          // 2) 当前块不再作为块头
            list_del(&(base->page_link));     // 3) 从链表中删除当前块头结点
            base = p;                         // 4) 将“当前块头”回退到前块，便于继续向右合并
        }
    }
```

```
47
48          // 向右合并
49          le = list_next(&(base->page_link));
50          if (le != &free_list) {
51              p = le2page(le, page_link);
52              if (base + base->property == p) {
53                  base->property += p->property;
54                  ClearPageProperty(p);
55                  list_del(&(p->page_link));
56              }
57          }
58      }
```

3. 算法是否有进一步的改进空间?

目前算法查找的时间复杂度基于空闲块的数量，也就是`O(#blocks)`，当空闲块 很多的时候，best-fit每次分配都要全表扫描。对于这个算法，可以使用Buddy System(伙伴系统)进行改进，它可以将可管理的内存区域划分为2的幂次大小的块，并在分配与释放时以“伙伴”的方式进行拆分与合并。

伙伴系统相当于把整个内存空间分为了多层，先设置好一个基本块的大小，每一个阶层都是2的幂次方，依次递增。在查找可用块的时候，如果需要N大小的内存，只用查找`log N`次就能找到了，这样分配内存的时间复杂度就降到`O(log N)`了，但是这种方法需要重新分配空闲内存，相对更为复杂。

```
xyh@TaroHan:~/lab2$ make grade
>>>>>>>>>> here_make>>>>>>>>>>>
gmake[1]: Entering directory '/home/xyh/lab2' + cc kern/init/entry.S + cc kern/init/init.c + c
c kern/libs/stdio.c + cc kern/debug/panic.c + cc kern/driver/console.c + cc kern/driver/dtb.c
+ cc kern/mm/best_fit_pmm.c + cc kern/mm/default_pmm.c + cc kern/mm/pmm.c + cc libs/printfmt.c
 + cc libs/readline.c + cc libs/sbi.c + cc libs/string.c + ld bin/kernel riscv64-unknown-elf-o
bjcopy bin/kernel --strip-all -O binary bin/ucore.img gmake[1]: Leaving directory '/home/xyh/l
ab2'
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=986
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
Total Score: 25/25
```

扩展练习Challenge：buddy system（伙伴系统）分配算法

在uCore中实现伙伴系统的页分配器，块的大小只取2的幂（以页为基本单位），在ucore中最小分配单位为页，这里每一页大小固定为4KB，使用伙伴系统进行2的幂次分配指的就是一次分配**2^n 页**。使得

分配内存的时间复杂度为 **O(log N)**，支持快速合并，外部碎片低。相关设计文档位于代码文件开头处。

1. 关键思路

该实现的核心思想参考了《伙伴分配器的一个极简实现》的结构设计。整个内存的可用区域被抽象为一棵**完全二叉树**，根节点表示整个管理范围的内存页，每次分配或释放操作都通过在这棵树上更新节点信息来完成。树中的每个节点都记录其对应内存区域中**最大的连续空闲块大小**，这种信息自底向上保持一致，保证分配时能够快速定位到合适的内存块。

当有新的内存请求到达时，分配器会将请求的页数通过 `fixsize()` 函数向上取整为最接近的 2 的幂值，例如请求 3 页则分配 4 页，以保证块大小始终符合二叉树划分结构的约束。接着，算法从根节点开始，沿着树结构向下搜索最小的可满足请求的节点。如果左子树中有足够大的空闲块，则进入左子树；否则进入右子树。最终找到目标块后，将该节点标记为已分配状态（`longest[index] = 0`），并向上更新父节点的 `longest` 值，使每个上层节点始终反映当前层级下的最大空闲块信息。这一过程只需沿树路径移动，时间复杂度为 O(log N)。

在释放阶段，算法根据要释放的物理页计算其在伙伴系统中的偏移，定位到对应叶子节点。随后将该节点标记为空闲，并自底向上检查其“伙伴”节点是否也为空闲。如果左右两个子节点的空闲块大小之和等于父节点的块大小，就意味着这两个块可以合并成一个更大的空闲块。系统在合并后更新父节点的 `longest` 值，并继续向上递归，直到无法再合并为止。通过这种方式，伙伴系统能在内存碎片较多的情况下自动恢复较大的连续空间，避免外部碎片积累。整个机制在设计上非常简洁，只依靠一个整数数组 `longest[]` 来记录整棵树的状态，而无需复杂的链表结构。每次分配与释放操作都局限在树的局部更新中，避免全局扫描，提高了效率。

综上，伙伴系统的关键思路可以概括为：**用一棵完全二叉树描述可管理的物理内存空间，每个节点记录最大可用块大小，分配时自顶向下选择最小满足请求的节点并标记占用，释放时自底向上检查并合并伙伴块，以保持最大空闲块连续性。** 这种结构既能实现高效的 O(log N) 分配与回收，又能自动管理碎片。

2. buddy_alloc（被 buddy_alloc_pages 调用）

以下为 `buddy_alloc` 的原样片段

代码块

```
//传入alloc_size,分配内存块，返回块在管理范围内的页偏移
static int buddy_alloc(unsigned size) {
    unsigned index = 0;
    unsigned node_size;
    unsigned offset = 0;

    if (size <= 0)
        size = 1;
    else if (!IS_POWER_OF_2(size))
```

```
        size = fixsize(size);  // 非2的幂大小自动对齐

    // 检查是否有足够大的空闲块
    if (buddy_mgr.longest[index] < size)
        return -1; // 根节点都没有足够大的块，直接失败

    // 从根节点开始搜索，找到最小的满足需求的块的index，退出时
node_size=size(alloc_size)
    for (node_size = buddy_mgr.size; node_size != size; node_size /= 2) {
        if (buddy_mgr.longest[LEFT_LEAF(index)] >= size) {
            index = LEFT_LEAF(index);  // 左子树有合适块，走左路
        } else {
            index = RIGHT_LEAF(index); // 右子树有合适块，走右路
        }
    }

    // 标记该块为已分配（该节点最大空闲块大小设为0）
    buddy_mgr.longest[index] = 0;
    offset = (index + 1) * node_size - buddy_mgr.size;  // 计算其在管理范围内的页
偏移页偏移

    // 向上更新父节点的最长空闲块大小
    while (index != 0) {
        index = PARENT(index);
        buddy_mgr.longest[index] = MAX(
            buddy_mgr.longest[LEFT_LEAF(index)],
            buddy_mgr.longest[RIGHT_LEAF(index)]
        );
    }

    return offset;
}
```

2.1 对齐到 2 的幂

因为伙伴系统的块大小必须是 2 的幂，所以这里先把用户请求向上对齐，保证后续可以按层对半拆分。

> 注：这里其实是双重保险，从 buddy_alloc_pages 传入前已经处理过一次。

代码块

```
if (size <= 0)
    size = 1;
else if (!IS_POWER_OF_2(size))
    size = fixsize(size);  // 非2的幂大小自动对齐
```

2.2 可用性快检

代码块

```
// 检查是否有足够大的空闲块
if (buddy_mgr.longest[index] < size)
    return -1; // 根节点都没有足够大的块，直接失败
```

`longest[0]` 是根节点覆盖的区域内“当前可用的**最大**连续空闲块大小”。如果连全局最大连续块都小于请求，就不可能成功——直接失败，以**O(1)** 快速剪枝，不需要向下检索。

2.3 逐层二分：自顶向下查找最小可用块

`node_size` 从根节点管理的页数开始，每下一层对半减小，直到正好等于目标块大小 `size`。这就保证走到的节点大小恰好等于请求的对齐后大小。

代码块

```
// 从根节点开始搜索，找到最小的满足需求的块的index，退出时node_size=size(alloc_size)
for (node_size = buddy_mgr.size; node_size != size; node_size /= 2) {
    if (buddy_mgr.longest[LEFT_LEAF(index)] >= size) {
        index = LEFT_LEAF(index);  // 左子树有合适块，走左路
    } else {
        index = RIGHT_LEAF(index); // 右子树有合适块，走右路
    }
}
```

这里采用左优先策略，每层判断左孩子覆盖区域是否存在足够大的空闲块；能满足则走左，否则走右。

用二叉树而不是链表，就不需要显式的拆分子块。只要沿路选择子树并最终把目标节点 `longest` 置 0，就等价于把路径上用到的更大块拆分为两个伙伴块，然后选择其中一个。 `longest` 的逐层回填会让父节点反映剩余子树中的最大空闲块大小。

2.4 标记占用 计算偏移

代码块

```
// 标记该块为已分配（该节点最大空闲块大小设为0）
buddy_mgr.longest[index] = 0;
offset = (index + 1) * node_size - buddy_mgr.size;  // 计算其在管理范围内的页偏移页偏移
```

到达目标节点后，将该节点的最大空闲块 `longest[index]` 设为 0，表示**这块完全被占用**。

`Offset` 是物理内存块在整个管理范围内的页偏移，即第 `index` 个节点对应的块的起始位。`index` 是当前节点在 `longest[]` 数组中的下标； `node_size` 是当前节点代表的内存块大小；`buddy_mgr.size` 是整个伙伴系统管理的页数。

2.5 自底向上回填longest

虽然只改动了一个叶层节点，但这会影响它所有祖先节点的**最大可用块** `longest[index]` 的统计。于是应该一路回溯：父节点的 `longest` = 左右子树 `longest` 的**最大值**。

代码块

```
while (index != 0) {
    index = PARENT(index);
    buddy_mgr.longest[index] = MAX(
        buddy_mgr.longest[LEFT_LEAF(index)],
        buddy_mgr.longest[RIGHT_LEAF(index)]
    );
}
```

3. buddy_free（被 buddy_free_pages 调用）

以下为 `buddy_free` 的原样片段。函数的输入就是offset变量，即页偏移。

代码块

```
// 释放内存块，根据页偏移找到对应块，自动合并相邻伙伴块
static void buddy_free(int offset) {
    unsigned node_size, index = 0;
    unsigned left_longest, right_longest;

    // 校验偏移合法性
    assert(offset >= 0 && offset < buddy_mgr.size && "buddy_free: invalid offset");

    // 从叶子节点开始向上查找，确定块大小
    node_size = 1;
    index = offset + buddy_mgr.size - 1;  // 转换为叶子节点索引

    // 找到对应的节点（确保该节点是已分配状态，即退出时longest[index]为0）
    for (; buddy_mgr.longest[index]; index = PARENT(index)) {
        node_size *= 2;
        if (index == 0)  // 到达根节点仍未找到，直接返回
            return;
    }

    // 标记该块为空闲，也就是设置页数
    buddy_mgr.longest[index] = node_size;
```

```
22
23      // 向上合并伙伴块
24      while (index != 0) {
25          index = PARENT(index);
26          node_size *= 2;  // 父节点的块大小是当前节点的2倍
27
28          left_longest = buddy_mgr.longest[LEFT_LEAF(index)];
29          right_longest = buddy_mgr.longest[RIGHT_LEAF(index)];
30
31          // 若左右子节点均空闲且大小之和等于父节点大小，则合并
32          if (left_longest + right_longest == node_size) {
33              buddy_mgr.longest[index] = node_size;
34          } else {
35              // 否则，父节点的最大空闲块为左右子节点的最大值
36              buddy_mgr.longest[index] = MAX(left_longest, right_longest);
37          }
38      }
39  }
```

3.1 入口检查

我们用 `assert` 语句来检查offset的基本边界，释放块的起点必须落在管理范围 `[0, size)` 内，如果不在，返回invalid报错。

3.2 自叶向上找第一个 `longest == 0`

代码块

```
1   node_size = 1;
2   index = offset + buddy_mgr.size - 1;  // 叶子索引
3   for (; buddy_mgr.longest[index]; index = PARENT(index)) {
4       node_size *= 2;
5       if (index == 0) return;
6   }
```

我们首先把页偏移映射到叶子节点索引。因为完全二叉树顺序存储下，叶子层的第 0 个叶子在数组索引 `buddy_mgr.size - 1`，所以第 `offset` 个叶子的索引是 `offset + buddy_mgr.size - 1`。

而下面的循环，意味着从该叶子所在位置向上找**第一个** `longest[index] == 0` 的节点。释放时，从叶子往上爬，**遇到的第一个 0** 就是当初分配的最高的那一层的节点，也就是块头节点。每向上一层，`node_size` *= 2，对应块大小翻倍；当我们跳出循环时，`node_size` 恰好就是这次释放块的原始块大小。

如果一路往上 `longest[index]` 都非 0，直到根也没遇到 0，说明这条路径上**没有被标记为占用的块**。这说明可能是重复释放或参数错误。此时代码直接 `return`，是“安全失败”。

3.3 把该块标记为空闲

代码块

```
buddy_mgr.longest[index] = node_size;
```

释放块的核心动作就是把这层节点的 `longest` 从 0 改为它的块大小，这意味着“这片区域现在出现了一个**完整可用**的块。到这一步为止，我们只是把**这一层**的局部状态改对了。整棵树其他节点还没有同步，需要向上合并和回填。

3.4 向上尝试“伙伴合并”

代码块

```
while (index != 0) {
    index = PARENT(index);
    node_size *= 2;

    left_longest  = buddy_mgr.longest[LEFT_LEAF(index)];
    right_longest = buddy_mgr.longest[RIGHT_LEAF(index)];

    if (left_longest + right_longest == node_size) {
        buddy_mgr.longest[index] = node_size;      // 完全合并成父块
    } else {
        buddy_mgr.longest[index] = MAX(left_longest, right_longest); // 只能取最大可用块
    }
}
```

我们通过**子结点求和==父块大小**来判断可合并。父节点覆盖的区域大小是 `node_size`，如果左右子树的“最大空闲块”分别是 `A、B`，且 **`A + B == node_size`**，就意味**左右子树各自都完全空闲**，它们可以合并成一个更大的完整父块。

反之，如果左右子树中任何一边被切碎或部分占用，就不可能合成完整父块，只能把父节点的 `longest` 设为两边“最大空闲块”的较大者，维持“不变量：该区域内可用的最大连续块大小”。

4. buddy_check

以下日志展示了伙伴系统初始化完成后，通过四大核心场景测试的全过程，所有测试项均符合预期，证明伙伴系统物理内存管理功能完整、逻辑正确。

代码块

```
buddy_pmm: 初始化完成
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
```

```
  memory: 0x0000000004000000, [0x0000000080367000, 0x0000000084366fff].
buddy: 已初始化 16384 页 (基地址物理地址: 0x0000000080367000)
[buddy_check] begin
管理页数: 16384 页，物理基地址: 0x0000000080367000

=== [1] 基础功能测试: 1页 + 3页 + 4页 分配释放 ===
  [Step] 初始化状态: 空闲页数=16384, 根节点最大块=16384
Step1: Alloc 1 page at 0x0000000080367000
  [Step] 分配1页: 空闲页数=16383, 根节点最大块=8192
Step2: Alloc 3 pages (aligned to 4) at 0x000000008036b000
  [Step] 分配3页（对齐4页）: 空闲页数=16379, 根节点最大块=8192
Step3: Freed 1 page at 0x0000000080367000
  [Step] 释放1页: 空闲页数=16380, 根节点最大块=8192
Step4: Alloc 4 pages at 0x0000000080367000
  [Step] 分配4页: 空闲页数=16376, 根节点最大块=8192
buddy: warning: 释放参数n=3与实际块大小4不匹配，使用实际大小4
Step5: Freed 3-page block (4 aligned) at 0x000000008036b000
  [Step] 释放3页块: 空闲页数=16380, 根节点最大块=8192
Step6: Freed 4-page block at 0x0000000080367000
  [Step] 释放4页块: 空闲页数=16384, 根节点最大块=16384
 基础功能测试通过: 1页+3页+4页 测试完成

=== [2] 边界场景测试: 分配满后再试1页 ===
  已分配最大页数: 16384 页，起始物理地址: 0x0000000080367000
  [Step] 分配最大页数: 空闲页数=0, 根节点最大块=0
buddy: 分配 1 页失败 (无空闲块)
  在满内存情况下再分配 1 页: 返回 NULL (正确)
  [Step] 释放最大页数: 空闲页数=16384, 根节点最大块=16384
  边界场景测试通过: 分配满→再分配失败→释放后恢复

=== [3] 合并逻辑测试: 完整合并+部分合并 ===
  [子测试1] 无阻碍合并（合并至根节点）
  [Step] 已合并至根节点（无阻碍）: 空闲页数=16384, 根节点最大块=16384
  [子测试2] 有阻碍合并（受阻层级停止）
  [Step] 已合并至8192页（被阻碍块拦截）: 空闲页数=8192, 根节点最大块=8192
  合并逻辑测试通过

=== [4] 异常场景测试 ===
  超量分配: 正确返回NULL
  释放非起始页: 已识别（页偏移=2，块大小=4） → 正确
  异常场景测试通过
all cases passed
check_alloc_page() succeeded!
```

4.1 初始化阶段：确认管理范围与基础状态

初始化是后续测试的前提，日志中关键信息表明伙伴系统已正确对接物理内存。

输出显示，实际管理 16384 页 物理内存，物理基地址为 0x80367000。系统可用物理内存区间是 [0x80200000, 0x88000000] ，其中0x80367000前是OS内核+页表两部分的内容。我们从 0x80367000 开始，避开了内核自身占用的内存。

打印 “buddy_pmm: 初始化完成” 和 “buddy: 已初始化 16384 页” ，说明页描述符元数据、二叉树 longest 数组均已完成初始化，可进入测试阶段。

4.2 四个测试场景

1. 基础功能测试：验证常规分配释放逻辑

- 测试目的：验证单页分配与释放；非 2 的幂（3 页）分配的自动对齐机制；释放后空闲页能否被再次复用；空闲块合并是否正常，系统最终能恢复到初始状态。
- 关键结果解读：

  Step1：可以看到，分配 1 页后，空闲页数从 16384 变为了 16383，根节点最大块从 16384 变为了 8192，这是因为 16384 页块被切割为两个 8192 页块，分配 1 页后剩余一个 8192 页空闲块。 物理地址 0x80367000，说明从物理内存管理的最开始位置开始分配，左优先规则成立。

  Step2：接下来请求 3 页时自动对齐为 4 页，分配后空闲页数从 16383 减少到了 16379，这说明3页请求可以自动向上对齐为 4 页。分配的起始地址是0x8036B000，离物理内存起点间隔4页大小，这正是因为前4页节点已经有一页分配，最大可分配页数变为3，于是开始找右侧的兄弟节点，正好是4页可分配，可满足。

  Step3：释放前面的 1 页，观察空闲页数可知，释放单页成功。0x80367000已经被释放，之后应该触发合并逻辑，向父节点更新longest数组的大小。

  Step4：再分配4页，目的是观察之前的释放与合并操作是否正常，能否利用之前分配过的块。新的 4 页分配的起始物理地址是0x80367000，说明**复用了**刚刚释放的空闲区域，说明伙伴系统能正确寻找最小可用块。

  Step5：系统提示“释放参数与块大小不符”但自动纠正为4页，释放成功。该警告验证了系统能容忍轻微调用错误。

  Step6：释放最后的4块。释放完成后，空闲页与最大块均恢复初始值，表明合并逻辑正常。
- 测试结果：常规分配释放、自动对齐逻辑均正常，无内存泄漏。

2. 边界场景测试：验证极端内存操作

- 测试目的：验证在极端内存使用情况下，伙伴系统是否能正确分配出全部可用页；在没有空闲页时，拒绝新的分配请求并返回 `NULL`
- 关键结果解读：

  Step 1：分配最大页数：16384 页，空闲页数=0，根节点最大块=0，说明所有页都被占用。`longest[0]=0` 说明已无可用连续块。、

  Step 2：尝试再分配 1 页，输出：`buddy: 分配 1 页失败 (无空闲块)`；返回值 `NULL`。这表明系统正确检测“无空闲块”的状态，没有产生越界或非法分配。

Step 3：完整释放，空闲页数恢复为16384，根节点最大块=16384。释放后，伙伴系统重新合并为一个大块，状态完全恢复，说明合并逻辑无误。

- 测试结果：极端内存操作无异常，全量内存分配释放后能完全恢复初始状态，无残留。

3. 合并逻辑测试：验证伙伴系统核心能力

- 测试目的：合并是伙伴系统的核心优势，需验证无阻碍时合并至根节点和有阻碍时停止合并两种场景，确保合并逻辑不越界、不遗漏。关键结果解读：
- 关键结果解读：

  无阻碍合并：先分配两块互为伙伴的小块，再依次释放后，它们成功沿树向上逐层合并，**一直合并到根**。空闲页数恢复 16384、根节点最大块恢复 16384，证明两个 2 页块合并为 4 页→8 页→…→最终合并为完整的 16384 页根节点。

  有阻碍合并：先用半区大块（8192页）占住根的一半作为阻碍，再在另一半分配两个 2 页块并释放。结果显示最大空闲块只能**合并到半区顶部（8192页）**，被阻碍块拦住，不会跨越阻碍继续合到根。释放后，空闲页数 = 8192、根节点最大块 = 8192。符合仅合并空闲伙伴块的设计逻辑。

- 测试结果：合并逻辑无异常，无阻碍时能完整合并至根节点，有阻碍时能正确停止，无错误合并。

4. 异常场景测试：验证错误拦截能力

- 测试目的：验证系统对非法请求的拦截能力，避免因异常输入导致崩溃。

  超量分配：请求分配 16384+1=16385 页（超过管理上限），系统正确返回 NULL，无崩溃。

  释放非起始页：尝试释放 4 页块的中间页（页偏移 = 2，块大小 = 4），系统能识别非起始页并打印 “正确”，且仅释放合法的起始页，无 panic。

- 测试结果：异常输入拦截逻辑有效，能抵御非法操作，系统稳定性达标。

4.3 最终结论：测试全通过，功能正常

日志末尾打印 all cases passed 和 check_alloc_page() succeeded!

伙伴系统的分配、释放、合并核心功能均正常，边界场景、异常场景的处理无漏洞。

能正确对接系统内存映射，可作为 ucore 的物理内存管理器正常工作。

扩展练习Challenge：任意大小的内存单元slub分配算法

1. 设计目标

实现一个两层架构的内存分配算法。第一层是页分配器，分配固定大小的页；第二层是对象分配器，需要在页上划分出多个固定大小的小块，将一页划分成多个64B/128B/256B等等大小的对象。因此，SLUB的机制就是会在页内部再划分小单元，供小对象分配使用。本实现是一个极简版的 SLUB（Slab Allocator），旨在在现有 buddy system 的基础上，提供高效的小对象分配机制。

目标：

1.1 减少频繁向 buddy 申请页的开销

1.2 在页内进行固定大小对象分配

1.3 保持实现简单，便于测试与教学

1.4 大对象仍走 buddy system

2. 总体结构

为了简化slub的实现，我们设计固定的比较小的槽位，这里设定为64B，128B，256B三级。当申请的内存空间小于64B的时候，就使用slub算法分配出连续的64B小内存空间使用；申请的内存空间大于64B小于128B时，就使用slub算法分配出连续的128B小内存空间使用；申请的内存空间大于128B小于256B时，就使用slub算法分配出连续的256B小内存空间使用；当申请的空间大于256B的时候，直接申请一整个页供于使用，这里的页分配机制使用已经写好的buddy system算法。这里设计的系统中共有两层分配机制：

层级	说明	分配粒度	对应函数
SLUB 层	管理小对象（≤ SLUB_OBJ_SIZE）	固定槽位（在这里设定为 64B,128B，256B）	slub_alloc / slub_free
Buddy 层	管理页级及以上的内存	页（4KB）	alloc_pages / free_pages

当用户通过 kmalloc(size) 申请内存时：

- 若 size ⩽ SLUB_OBJ_SIZE，走 SLUB；
- 若 size > SLUB_OBJ_SIZE，走 Buddy；
- kfree() 自动识别来源（SLUB 或 Buddy）。

3. 主要数据结构

3.1 Slab

每个 slab 对应一页内存，包含若干固定大小对象槽位。

代码块

```
ypedef struct Slab {
    list_entry_t link;    // 链入全局 slab_list
    struct Page *page;    // 对应页
    size_t free_cnt;      // 当前空闲对象数量
    size_t objs;          // 总对象数量
```

```
    unsigned magic;       // 用于检测 slab 有效性
} slab_t;
```

页内布局：

| slab_t头(结构体) | 对象区(Object[]) | 位图(bitmap[]) |

3.2 BigAllocHeader

用于标识超过 SLUB 阈值的大块分配（页路径）。

代码块

```
typedef struct BigAllocHeader {
    uint32_t magic;       // 用于识别
    uint32_t npages;      // 占用页数
    struct Page *first;   // 首页指针
} BigAllocHeader;
```

4. 关键参数

- SLUB_OBJ_SIZE：SLUB 管理的对象大小
- SLUB_ALIGN：对齐粒度（16B）
- SLUB_MAGIC：slab校验
- BIG_MAGIC：大块页分配

5. 内存布局与对齐

- 页首放slab_t
- 对象区起始于sizeof(slab_t)之后
- 位图放在页末
- 对象区按SLUB_ALIGN对齐
- 每个对象大小固定为SLUB_OBJ_SIZE
- 页内对象数由 compute_objs_per_slab()自动计算

6. 分配与回收流程

6.1 kmalloc(size)

代码块

```
if (size <= SLUB_OBJ_SIZE)
    return slub_alloc(size);
else
    return buddy //分配 (多页);
```

超过 SLUB 阈值的对象由 buddy system 提供整页分配。

6.2 slub_alloc(size)

遍历 slab_list；找到 free_cnt > 0 的 slab；使用位图分配一个空槽；若找不到，则通过 new_slab() 新建 slab；返回对象指针。

代码块

```
void *slub_alloc(size_t size) {
    if (size == 0 || size > SLUB_OBJ_SIZE) return NULL;
    (void)align_up(size, SLUB_ALIGN);

    // 1) 在现有 slab 中找空闲
    list_entry_t *le = &slab_list;
    while ((le = list_next(le)) != &slab_list) {
        slab_t *s = le2slab(le);
        if (s->free_cnt == 0) continue;
        void *p = slab_alloc_from(s);
        if (p != NULL) return p;
    }
    // 2) 新建 slab
    slab_t *s = new_slab();
    if (!s) return NULL;
    return slab_alloc_from(s);                  // 新 slab 第一次分配一定成功
}
```

6.3 slub_free(obj)

根据对象地址找到所在页（页对齐运算）；验证页首 slab_t.magic；清除对应位图 bit；若 slab 全空，则回收整页给 buddy。

代码块

```c
void slub_free(void *obj) {
    if (!obj) return;
    // 通过“页对齐”直接定位 slab 头，避免全表扫描
    void *kva_base = (void *)((uintptr_t)obj & ~(PGSIZE - 1));
    slab_t *s = (slab_t *)kva_base;

    // 简单健壮性检查
    if (s->magic != SLUB_MAGIC) {
        cprintf("slub_free: bad obj %p (magic)\n", obj);
        return;
    }
    // 边界检查（保证 obj 落在对象区内、且是 64B 对齐）
    char *base = slab_objs_base(s);
    size_t off = (size_t)((char *)obj - base);
    if (off >= s->objs * SLUB_OBJ_SIZE || (off % SLUB_OBJ_SIZE) != 0) {
        cprintf("slub_free: bad obj %p (range/alignment)\n", obj);
        return;
    }

    slab_free_to(s, obj);

    // slab 全空时直接归还整页
    if (s->free_cnt == s->objs) {
        list_del(&s->link);
        free_pages(s->page, 1);                 // 归还给 buddy
    }
}
```

6.4 kfree(ptr)

判断指针是否带有 BigAllocHeader.magic == BIG_MAGIC；若是 → 释放整页；否则 → 调用 slub_free()。

代码块

```c
void kfree(void *ptr) {
    if (!ptr) return;

    // 先尝试按“页分配”路径释放
    BigAllocHeader *h = (BigAllocHeader *)((char *)ptr -
sizeof(BigAllocHeader));
    if (h->magic == BIG_MAGIC && h->first && h->npages > 0) {
        free_pages(h->first, h->npages);
        return;
    }
```

```
    // 否则当作 SLUB 小对象
    slub_free(ptr);
}
```

7. 自检机制

- slub_selftest() 自动验证
- 小对象分配（≤ SLUB_OBJ_SIZE）
- 多对象在同一页的分布
- 超过 SLUB 阈值的对象走页路径
- 页对齐校验
- SLUB 与 buddy 地址区分验证
- 全部通过后打印

8. 实验结果check()

运行check代码，结果如下：

代码块

```
check_alloc_page() succeeded!
SLUB(level 0): obj=64B, align=64B
SLUB(level 1): obj=128B, align=128B
SLUB(level 2): obj=256B, align=256B

[slub_selftest] begin
L1 objs: 0xffffffffc0368030 (32B→64B), 0xffffffffc0368070 (64B)
L2 objs: 0xffffffffc0369030 (80B→128B), 0xffffffffc03690b0 (128B)
L3 objs: 0xffffffffc036a030 (160B→256B), 0xffffffffc036a130 (256B)
Big obj: 0xffffffffc036b010 (512B), pages=1
已释放所有对象

验证相同大小的块地址在一起:
分配顺序: a1(64B), b1(256B), a2(32B), c1(128B), b2(130B), c2(100B)
a1 = 0xffffffffc0368030 (64B)
b1 = 0xffffffffc0369030 (256B)
a2 = 0xffffffffc0368070 (64B again)
c1 = 0xffffffffc036a030 (128B)
b2 = 0xffffffffc0369130 (256B again)
c2 = 0xffffffffc036a0b0 (128B again)
Reuse L1 obj: 0xffffffffc03680b0 (64B)
```

```
[slub_selftest] all ok

satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
```

首先我们关注第一部分的分配。每个大小不同的块都会申请到最接近大小的块，例如32B会申请到64B块，80B申请到128B块。相同级别的对象地址相邻且等距排列，符合页内“线性切片”设计。不同级别（64B / 128B / 256B）的地址分布在不同页段中，间隔明显。

然后我们进行跨级分配验证，目的就是为了验证地址是怎么分配的。实际上，每个等级维护自己的 slab 列表，不同大小的对象不会混在同一页里。即使是按照不同的顺序进行申请内存，不同大小的对象的地址也不会相邻。一整页 4KB 只服务一个等级（同一大小的对象）。不会出现一半放 64B 对象，一半放 128B 对象的情况。

观察下一块的实验结果，我们发现的确如此，64B的a1和256B的b1申请出来的地址隔着4K的一个页大小。之后再申请64B的a2，他会在a1后64B地址空间处摆放。这也验证了相同大小的块地址在一起摆放。

扩展练习Challenge：硬件的可用物理内存范围的获取方法

> 如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围?

已知，在之前的实验中，OS 是通过设备树获取可用物理内存范围的。

OS 本身不直接探测硬件内存，而是依赖 Bootloader（OpenSBI） 先完成硬件探测，再通过设备树把内存信息告诉内核。

OpenSBI 在启动阶段会：

- 扫描物理内存控制器，确定硬件实际存在的物理内存范围。
- 将这些内存信息写入设备树的 /memory 节点中。
- 把 DTB 的物理地址作为参数，传递给内核。

内核验证 DTB 的合法性后，遍历 DTB 找到/memory子节点。最后，解析/memory节点的reg属性，提取出物理内存的起始地址和总大小。而page_init()函数中调用的get_memory_base()和get_memory_size()，本质就是封装了上述解析reg属性的过程，分别返回内存基址和大小。

那么，如果 OS 无法通过设备树或 BIOS 等外部帮助获取内存范围，就需要内核自己实现物理内存探测逻辑，即主动测试哪些物理地址是可读写的（即属于 DRAM），哪些是不可用的。

解决方案

核心目标是获取 **物理内存基址（mem_begin）** 和 **总大小（mem_size）**。这两个是唯一需主动获取的源头数据，其他如内存结束地址、空闲区范围等均可通过计算得出。

利用 Qemu virt 固定布局硬编码

- 硬编码默认源头数据

mem_begin：直接使用 DRAM_BASE 宏（0x80000000）。这是 Qemu virt 平台对 DRAM 起始地址的强制约定。

默认 mem_end：直接使用 PHYSICAL_MEMORY_END 宏（0x88000000）。

默认 mem_size：按 PHYSICAL_MEMORY_END - DRAM_BASE 计算，即 0x88000000 - 0x80000000 = 0x8000000（128MB）。此值为 Qemu 未通过 -m 参数修改内存大小时的默认容量，由平台预先定义。

- 硬件验证

验证的核心目的是检测用户是否使用 -m 修改了内存大小，通过异常捕获和地址递增或递减测试的方式定位实际内存边界，避免纯硬编码的缺陷。

在进行地址测试前，需提前通过 RISC-V 的 stvec 寄存器配置自定义异常处理函数。当检测到异常时，它会标记当前测试地址为无效地址，并通过调整程序计数器跳回测试循环的下一次迭代，让地址递增或递减测试能继续进行。

  - 实际内存比默认小

  向默认 mem_end （0x88000000）写入校验值会立即触发访问异常。此时需从默认 mem_end 开始，按4KB页为单位向 mem_begin 递减测试。每次向当前地址写入校验值，若触发异常则继续递减，直到某地址未触发异常，且读取数据与写入值一致（确认是可读写DRAM）。该地址即为实际内存的最后一个有效页地址，向上对齐到4KB页边界后，得到真实 mem_end ，并以此计算实际 mem_size 。

  - 实际内存比默认大

  向默认 mem_end 写入校验值不会触发异常，且因为该地址仍在实际DRAM范围内，读取数据与写入值一致。此时需从默认 mem_end 开始，按4KB页为单位向上递增测试，持续向更高地址写入校验值，直到某地址触发访问异常。前一个未触发异常且读写一致的地址即为实际内存的最后一个有效页地址，向上对齐到4KB页边界后，得到真实 mem_end 。

  - 实际内存和默认一样大

  向默认 mem_end 写入校验值不会触发异常，且读取数据与写入值一致。此时从默认 mem_end 向上递增一页测试，写入校验值就会立即触发访问异常。这说明默认 mem_end 就是真实边界，无需调整参数，直接沿用 mem_begin=0x80000000 、 mem_end=0x88000000 、 mem_size=0x8000000 即可。

> 虽然从 Qemu virt 的物理内存布局扫描结果来看，外设、只读存储等非 DRAM 区域集中在 0x80000000之前，，但这并不意味着高地址段绝对不会出现非 DRAM 区域。比如用户通过自定义 Qemu 配置，在高地址段额外映射了调试模块、扩展外设。甚至 OpenSBI 初始化时，会在 DRAM 中预留部分特殊页，如用于中断处理的保留页，这些页面虽属于 DRAM 地址范围，却可能被设置为只

> 读或不可修改。若只通过访问是否触发异常判断，会误将这些非标准 DRAM 区域当成可用内存，后续写入数据时可能导致硬件功能异常或数据丢失，因此采用的是写入校验值后读取对比的方式。
>
> 当测试中遇到访问不触发异常，但读取值与写入值不一致的情况时，需按以下逻辑处理：首先标记当前测试地址为非可用 DRAM 区域，避免后续内存管理使用该地址。接着继续按原方向测试下一个 4KB 页地址，若只是单页出现不一致，而后续页面恢复读写一致，则将该不一致页标记为内存空洞，跳过即可，不影响整体 DRAM 范围的判断。若连续多页均出现读写不一致，则说明当前地址已超出实际可用 DRAM 的范围，需以最后一个读写一致的地址作为边界，向上对齐到 4KB 页边界后确定真实 mem_end，确保最终获取的内存范围中，每一页都能稳定存储数据。

本实验中重要的知识点及与对应的OS原理中的知识点

- 页表管理中的分页操作，把内存划分为固定大小的页（4KB），可以实现高效映射；又比如伙伴系统把页组织成树状结构，按 2 的幂次划分。这些都是将地址空间分块化，把申请、释放、合并、分裂操作限制在相邻的同阶块中，大大简化了管理复杂度。这对应了OS中的核心思想：分块与层次化管理。大资源分层拆分，小资源按块组织，便于查找与回收。
- 实验在扩展练习中实现伙伴系统，用longest数组模拟完全二叉树，按 2 的幂次对齐分配内存并自底向上合并伙伴块，还通过基础、边界等多场景测试验证功能，这对应我们学到的 OS 原理是伙伴系统作为连续物理内存分配的高效算法，依托 2 的幂次块和二叉树管理实现 O (logN) 的分配与回收，同时能有效减少外部碎片。

OS原理中很重要，但在实验中没有对应上的知识点

- 实验未涉及缺页异常处理，仅聚焦物理内存分配回收，这对应我们学到的 OS 原理是缺页异常是虚拟内存的核心触发点，操作系统需通过保存上下文、判断异常原因、页换入、更新页表等步骤处理，以此实现内存的按需分配，提升内存利用率。