

# Lab2

许哲涵2311858 赵悦蛟2313650 赵宁2311576

## 一、实验目的

- 1、理解页表的建立和使用方法
- 2、理解物理内存的管理方法
- 3、理解页面分配算法

## 二、实验内容

### (一)、lab2中的操作系统知识点

#### 连续物理内存分配算法 (First - Fit、Best - Fit)

- **实验中知识点含义：**
  - First - Fit 算法：在空闲内存块链表中，从头开始查找，找到第一个能满足内存分配请求大小的空闲块，进行分配。
  - Best - Fit 算法：遍历所有空闲内存块，找到能满足请求大小且大小最接近请求大小的空闲块，进行分配。
- **OS 原理中知识点含义：**连续内存分配是操作系统为进程分配连续物理内存空间的方法，First - Fit 和 Best - Fit 是其中的经典算法。
- **关系：**实验中的算法实现是 OS 原理中连续内存分配算法的具体代码体现，原理指导实验的设计与实现。
- **差异：**原理层面对算法思想的抽象描述，关注算法的逻辑和性能特点（如 First - Fit 查找速度快但易产生碎片，Best - Fit 碎片少但查找开销大）；实验中则是通过代码将这些算法思想转化为可执行的功能，涉及具体的数据结构（如空闲块链表）和函数实现（如分配、释放函数）。

#### 页表的建立与使用

- **实验中知识点含义：**通过代码构建页表结构，实现虚拟地址到物理地址的映射，使得系统能基于页表进行地址转换，访问物理内存。
- **OS 原理中知识点含义：**页表是操作系统用于实现虚拟内存管理的重要数据结构，它存储了虚拟地址与物理地址的映射关系，辅助硬件（如 MMU）完成地址转换，实现虚拟内存到物理内存的映射，同时还能提供内存保护等功能。
- **关系：**实验是原理的实践，按照原理中页表的作用和结构设计，用代码实现页表的创建和地址映射过程。
- **差异：**原理更关注页表的整体架构、地址转换的逻辑流程以及内存保护等多方面的功能描述；实验则聚焦于在特定系统（如 ucore）中，如何通过具体的代码步骤来建立最简单的页表映射，实现基本的地址转换功能，对于复杂的内存保护等机制可能涉及较少。

#### 物理内存的管理方法（连续物理内存管理）

- **实验中知识点含义：**通过维护空闲块链表等数据结构，对物理内存进行分配和回收管理，确保内存的有效利用，主要针对连续的物理内存区域进行管理。
- **OS 原理中知识点含义：**物理内存管理是操作系统的核心功能之一，负责对系统中的物理内存进行分配、

回收、碎片整理等操作，以满足进程对内存的需求，连续物理内存管理是其中的一种方式（还有非连续等方式）。

- **关系**：实验是原理中连续物理内存管理的具体实践，依据原理中对物理内存管理的目标和方法，设计代码逻辑来管理物理内存。
- **差异**：原理涵盖了物理内存管理的各种策略、算法以及不同场景下的管理方式等广泛内容；实验则是在特定的实验环境下，实现连续物理内存管理的基本功能，规模和复杂度相对较低。

## (二)、OS 原理中重要但实验中未对应知识点

### 虚拟内存的置换算法（如 LRU、FIFO）

- **原理中重要性**：当系统内存不足时，需要将部分页面换出到外存，置换算法决定换出哪些页面，直接影响系统的性能，是虚拟内存管理的关键部分。
- **实验未对应原因**：实验主要聚焦于物理内存的分配和简单的页表映射，尚未涉及虚拟内存的页面置换场景，重点在内存的分配而非页面的置换管理。

### 内存的分段管理

- **原理中重要性**：分段管理是内存管理的一种重要方式，它将程序按照逻辑段（如代码段、数据段、堆栈段等）进行划分，便于程序的共享和保护，能更好地满足程序的逻辑结构需求。
- **实验未对应原因**：实验重点在连续物理内存分配和页表相关的分页管理，没有涉及分段管理的相关内容，主要围绕分页机制展开。

### 内存保护机制（如权限控制、访问验证）

- **原理中重要性**：内存保护能防止进程非法访问其他进程的内存或操作系统的内存，保障系统的稳定性和安全性，是操作系统内存管理不可或缺的部分。
- **实验未对应原因**：实验侧重于内存的分配和地址映射的实现，没有深入到内存保护的具体机制实现，如对不同内存区域设置访问权限等操作。

## 练习1：理解first-fit 连续物理内存分配算法

### 实现思路

想要实现最先匹配法(first-fit)就要按分区的先后次序，从头查找，找到符合要求的第一个分区就分配。该算法分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。

### 代码分析

#### 1. default\_init - 初始化管理器

```
static void default_init(void) {  
    list_init(&free_list); // 初始化空闲链表  
    nr_free = 0;           // 空闲页数量清零  
}
```

list\_init 函数

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

这段代码用来**初始化内存管理器的全局状态**。调用 `list_init` 函数，初始化一个空的双向链表 `free_list`，然后将空闲页数量清零。

## 2. default\_init\_memmap - 初始化内存映射

```
static void default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;

    //初始化每一页的属性
    for (; p != base + n; p++) {
        assert(PageReserved(p)); //确保页面是保留状态
        p->flags = p->property = 0; //清空标志位和属性
        set_page_ref(p, 0); //引用计数清零
    }

    //设置第一页为块头
    base->property = n; //记录整个空闲块的大小
    SetPageProperty(base); //标记此页为块头（有property属性）
    nr_free += n; //更新总空闲页数

    //将空闲块插入到有序链表中（按地址升序）
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link)); //链表为空直接插入
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            //找到插入位置：按地址从小到大排序
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link)); //插入到链表末尾
            }
        }
    }
}
```

这段代码的作用是**将一段连续的空闲页面初始化为一个空闲块，并按地址顺序插入链表**。

函数参数：

- `base`: 连续空闲页的起始页指针。
- `n`: 连续空闲页的数量，也就是我们需要进行初始化的页面数量。
- `p`: 用于遍历每个页面的指针。

首先, `assert(n > 0);` 进行断言检查。目的是确保传入的页面数量`n`是有效的, 可以防止传入0个或负数个页面这种无意义的参数。 `struct Page *p = base;` 将指针 `p` 初始化为指向传入的起始页面 `base`。

循环从头到尾遍历每一个页面, 首先通过断言检查确保当前页面确实是可用的保留内存, 防止操作错误的内存区域。然后清空页面的标志位和属性字段, 将页面恢复到原始的空白状态, 清除所有之前可能存在的状态信息。同时将页面的引用计数设置为零, 表示这些页面当前没有被任何进程或模块使用, **处于完全空闲可分配的状态**。通过这样的初始化, 这段连续内存中的所有页面都被重置为统一干净的初始状态, 为后续将这些页面作为一个完整的空闲内存块来管理奠定了基础。在循环结束后, 代码会单独设置第一页作为这个空闲块的块头, 记录整个块的大小信息, 而其他页面则保持简单的跟随状态。

然后, `nr_free += n` **更新了系统的总空闲页面计数**, 将新初始化的`n`个页面加入全局统计, 再根据空闲链表是否为空分别处理:

- 如果链表为空则直接将这个新空闲块插入链表;
- 如果链表非空则遍历整个链表, 通过比较页面地址来寻找合适的插入位置, 当找到第一个地址比新块大的现有块时就使用 `list_add_before` 函数在它前面插入, 或者如果遍历到链表末尾都没找到更大的块则使用 `list_add` 函数将其插入到链表尾部, 从而维护了整个空闲链表按物理地址从小到大排序的有序性。

```
static inline void
list_add(list_entry_t *listelm, list_entry_t *elm) {
    list_add_after(listelm, elm);
}
```

```
static inline void
list_add(list_entry_t *listelm, list_entry_t *elm) {
    list_add_after(listelm, elm);
}
```

### 3. `default_alloc_pages` - 分配页面

```
static struct Page * default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) return NULL; // 内存不足

    struct Page *page = NULL;
    list_entry_t *le = &free_list;

    // First-Fit: 查找第一个足够大的空闲块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) { // 找到合适的块
            page = p;
            break;
        }
    }

    if (page != NULL) {
        // 从链表中移除该块
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
    }
}
```

```

// 如果块大于需求，进行分割
if (page->property > n) {
    struct Page *p = page + n; // 剩余部分的起始页
    p->property = page->property - n; // 设置剩余大小
    SetPageProperty(p); // 标记为块头
    list_add(prev, &(p->page_link)); // 将剩余部分插回链表
}

nr_free -= n; // 更新空闲页计数
ClearPageProperty(page); // 清除块头标记（已分配）
}
return page;
}

```

这段代码的作用是**实现最先匹配算法，找到第一个足够大的空闲块进行分配，必要时进行分割。**

函数开始时，首先检查请求的页面数 `n` 是否合理，如果系统当前的总空闲页面数 `nr_free` 小于 `n`，说明物理内存根本不够用，直接返回 `NULL` 表示分配失败。然后通过遍历按地址排序的空闲链表，使用最先匹配算法寻找第一个大小属性 `property` 大于等于 `n` 的空闲块，这里的 `property` 字段记录着每个空闲块包含的连续页面总数。

当找到合适的空闲块后，需要将它从链表中移除，如果这个块的大小 `property` 正好等于请求的 `n`，就直接整个分配出去；但如果 `property` 大于 `n`，说明块比需求大，就需要进行分割：从第 `n` 个页面处将块切开，前面 `n` 个页面分配给请求者，剩下的 `property - n` 个页面作为一个新的空闲块，设置好它的 `property` 值并重新插入到链表中。最后更新系统的空闲页面计数 `nr_free`，将其减少 `n`，并清除已分配页面的块头标记，返回分配成功的起始页面指针。如果遍历完整个链表都没有找到满足大小 `n` 的空闲块，就返回 `NULL` 表示分配失败。

#### 4. default\_free\_pages - 释放页面

```

static void default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;

    // 初始化被释放的页面
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p)); // 验证状态
        p->flags = 0; // 清空标志
        set_page_ref(p, 0); // 引用计数清零
    }

    // 设置释放块的头信息
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    // 按地址顺序插入链表（与init_memmap类似的逻辑）
    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));
    }
}

// 向前合并：检查前一个空闲块是否能合并
list_entry_t* le = list_prev(&(base->page_link));
if (le != &free_list) {
    p = le2page(le, page_link);
    if (p + p->property == base) { // 地址连续
        p->property += base->property; // 合并大小
        clearPageProperty(base); // 取消当前块头
        list_del(&(base->page_link)); // 从链表移除当前块
        base = p; // 指向合并后的块
    }
}

// 向后合并：检查后一个空闲块是否能合并
le = list_next(&(base->page_link));
if (le != &free_list) {
    p = le2page(le, page_link);
    if (base + base->property == p) { // 地址连续
        base->property += p->property; // 合并大小
        clearPageProperty(p); // 取消后一块头
        list_del(&(p->page_link)); // 从链表移除后一块
    }
}
}
}

```

这段代码的作用是**释放已分配的页面，合并相邻的空闲块以减少碎片**。

首先，`base` 指向要释放的连续内存块的起始页面，`n` 表示要释放的连续页面数量。

函数开始时用指针 `p` 遍历这 `n` 个页面，确保它们不是保留页面且没有属性标志，然后将每个页面的标志位和引用计数清零，使它们回到初始状态。接着将 `base` 设置为这个释放块的块头，设置其 `property` 字段为 `n` 并标记为有空闲属性，同时更新全局空闲页面计数 `nr_free`。

随后函数将这块**新释放的内存按地址顺序插入空闲链表**，如果链表为空就直接插入，否则遍历链表找到合适的插入位置，维护链表的地址有序性。完成插入后，函数进行**关键的内存块合并操作**：首先检查前一个空闲块 `p`，如果 `p + p->property` 正好等于当前块 `base` 的地址，说明两块内存物理相邻，就将它们合并，清除 `base` 的块头标志并将其从链表中移除，让 `base` 指向前面的合并块；然后检查后一个空闲块 `p`，如果 `base + base->property` 正好等于后一块的地址，同样进行合并，更新 `base` 的大小属性并清除后一块的块头标志，将其从链表中移除。通过这两次合并操作，函数有效地将相邻的空闲内存块合并成更大的连续块，减少了内存碎片。

## 5、default\_nr\_free\_pages-获取空闲页面数函数

```
static size_t
default_nr_free_pages(void) {
    return nr_free;
}
```

函数体内直接返回全局变量 `nr_free`，这个 `nr_free` 在整个内存管理模块中维护着系统当前空闲页面的总数。每当通过 `default_alloc_pages` 成功分配页面时，`nr_free` 会减少相应的数量；而当通过 `default_free_pages` 释放页面时，`nr_free` 会增加相应的数量。因此**这个函数的作用就是提供一个接口，让外部调用者能够查询系统中当前可用的空闲页面总数**，它本身不进行任何计算或处理，只是简单地返回已经维护好的全局状态值。

## 6、basic\_check-检查函数

```
static void
basic_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);

    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));

    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    assert(alloc_page() == NULL);

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

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

    assert(alloc_page() == NULL);

    free_page(p0);
    assert(!list_empty(&free_list));
}
```

```

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

    assert(nr_free == 0);
    free_list = free_list_store;
    nr_free = nr_free_store;

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

```

这个 `basic_check` 函数的作用是对内存管理器的基本功能进行**自动化测试验证**，它通过一系列分配和释放操作来检查内存分配器是否正常工作，确保分配、释放、引用计数等基本机制没有错误。

函数中定义了三个页面指针 `p0`, `p1`, `p2` 用于测试，首先通过多次 `alloc_page()` 调用验证内存分配功能，确保能成功分配到不同的物理页面且这些页面的引用计数正确初始化。然后检查分配到的页面物理地址是否在有效范围内。接着函数保存当前的空闲链表状态和空闲页面计数，临时清空链表并设置空闲计数为零，验证在这种情况下分配页面会失败。随后释放之前分配的页面，确认空闲计数正确更新为3，再次分配页面并验证功能正常。

函数还特别测试了分配顺序的特性，释放 `p0` 后立即分配一个新页面，验证最先匹配算法会重新分配刚刚释放的页面。最后恢复之前保存的空闲链表状态和计数，并清理释放所有测试用的页面。整个测试过程通过断言来确保每个步骤都符合预期行为，如果任何检查失败则说明内存管理器实现存在缺陷。

## 7、default\_check-全面检查

```

static void
default_check(void) {
    int count = 0, total = 0;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        assert(PageProperty(p));
        count ++, total += p->property;
    }
    assert(total == nr_free_pages());

    basic_check();

    struct Page *p0 = alloc_pages(5), *p1, *p2;
    assert(p0 != NULL);
    assert(!PageProperty(p0));

    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));
    assert(alloc_page() == NULL);

    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    free_pages(p0 + 2, 3);
}

```



```

assert(alloc_pages(4) == NULL);
assert(PageProperty(p0 + 2) && p0[2].property == 3);
assert((p1 = alloc_pages(3)) != NULL);
assert(alloc_page() == NULL);
assert(p0 + 2 == p1);

p2 = p0 + 1;
free_page(p0);
free_pages(p1, 3);
assert(PageProperty(p0) && p0->property == 1);
assert(PageProperty(p1) && p1->property == 3);

assert((p0 = alloc_page()) == p2 - 1);
free_page(p0);
assert((p0 = alloc_pages(2)) == p2 + 1);

free_pages(p0, 2);
free_page(p2);

assert((p0 = alloc_pages(5)) != NULL);
assert(alloc_page() == NULL);

assert(nr_free == 0);
nr_free = nr_free_store;

free_list = free_list_store;
free_pages(p0, 5);

le = &free_list;
while ((le = list_next(le)) != &free_list) {
    struct Page *p = le2page(le, page_link);
    count --, total -= p->property;
}
assert(count == 0);
assert(total == 0);
}

```

`default_check` 函数的作用是对最先匹配算法内存管理器进行**全面而严格的集成测试**，它不仅验证基本功能，还专门测试了内存分割、合并、边界情况等复杂场景，确保整个内存管理系统的正确性和健壮性。

函数开始时使用 `count` 和 `total` 变量来**统计空闲链表中的块数和总页面数**，验证其与全局空闲计数的一致性。然后调用 `basic_check` 执行基础测试。接着通过分配大块内存 `p0` 测试复杂场景：临时清空空闲链表后，测试部分释放操作（释放 `p0 + 2` 开始的3个页面），验证分割后形成的空闲块属性正确，并检查内存合并机制——释放相邻块 `p0` 和 `p1` 后，测试它们能否正确合并成更大的连续空间。

函数还通过变量 `p2` 记录中间页面位置，测试分配算法在碎片环境下的行为，验证分配请求能否在合适的空闲位置得到满足。最后恢复原始的空闲链表状态，并再次验证统计信息的正确性。整个测试过程通过精心设计的分配释放序列和断言检查，全面验证了最先匹配算法在各种边界条件下的正确行为。

## 8、结构体 default\_pmm\_manager

```
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,
    .init_memmap = default_init_memmap,
    .alloc_pages = default_alloc_pages,
    .free_pages = default_free_pages,
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};
```

这个结构体的作用是**定义一个完整的内存管理模块**，它将之前实现的所有分散函数封装成一个统一的内存管理器接口。通过这样的封装，操作系统可以像使用插件一样使用不同的内存管理算法，只需要替换这个结构体实例就能切换不同的内存管理策略，大大提高了系统的模块化和可扩展性。

结构体中的每个字段都对应一个特定的功能：

- `name` 标识这个管理器为 "default\_pmm\_manager"
- `init` 指向初始化函数用于设置内存管理器的初始状态
- `init_memmap` 负责初始化物理内存映射
- `alloc_pages` 和 `free_pages` 分别处理页面的分配和释放请求
- `nr_free_pages` 提供当前空闲页面数的查询功能
- `check` 则用于执行管理器的自检测试

### 改进空间

1. **减少查找开销**：当前分配时需从链表头开始依次遍历空闲块，在空闲块数量较多时效率较低。可引入索引结构（如按块大小范围建立分级链表），分配时直接定位到可能满足需求的块所在链表，减少遍历次数。
2. **优化碎片管理**：虽然释放时会合并相邻块，但长期分配释放后仍可能产生大量小碎片。可增加碎片整理机制，如定期扫描链表，将连续的小空闲块合并为大块；或在分配大内存块失败时，主动尝试合并相邻小碎片后再分配。
3. **动态调整链表顺序**：当前空闲块按地址排序，可改为根据分配频率动态调整块在链表中的位置（如将近期分配过的块移至链表前端），提高常用块的查找速度。
4. **引入延迟合并策略**：释放内存时不立即合并相邻块，而是记录潜在合并信息，当分配请求失败或达到一定阈值时再执行合并，减少频繁合并带来的开销。
5. **支持块分割优化**：当前分配时若块大于需求则直接分割为两部分，可进一步考虑分割后剩余块的大小是否合理（如避免产生过小碎片），若剩余块过小可直接分配整个块，减少碎片数量。

## 练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

**Best-Fit实现思路**：在分配内存时遍历所有空闲块，找到大小最接近请求值的那个块而非第一个满足条件的块。

大部分地方和first-fit是一样的，因此我们只分析best-fit算法的核心部分：



### 三、Challenge

#### Challenge1: buddy system(伙伴系统)分配算法(需要编程)

##### (一)、相关文件作用说明

1. `buddy_system_pmm.h`

头文件，声明了伙伴系统内存管理器实例 `buddy_system_pmm_manager`，该实例实现了 ucore 操作系统中物理内存管理的接口规范，用于将伙伴系统算法注册为可用的内存分配策略。

2. `memlayout.h`

定义了伙伴系统的核心数据结构 `buddy_system_t`，该结构体用于管理所有空闲内存块的元信息，是伙伴系统运行的基础。

3. `buddy_system_pmm.c`

核心实现文件，包含伙伴系统的初始化、内存块分配、释放、分裂、合并等核心逻辑，以及辅助函数（如 2 的幂判断、阶数计算等）和调试函数（如显示空闲块信息）。

##### (二)、核心结构体说明

`buddy_system_t`（定义于 `memlayout.h`）

```
#define MAX_BUDDY_ORDER 14 // 0x7cb9 31929
typedef struct
{
    unsigned int max_order;           // 实际最大块的大小
    list_entry_t free_array[15];      // 伙伴堆数组
    unsigned int nr_free;             // 伙伴系统中剩余的空闲块
} buddy_system_t;
```

- **作用：**管理伙伴系统的全局状态。`free_array` 是核心，每个索引 `n` 对应一个链表，存储所有大小为  $2^n$  页的空闲块；`max_order` 限制了最大块的大小；`nr_free` 用于快速跟踪空闲页总数。

##### (三)、核心函数作用与目的

###### 1. 基础工具函数

`IS_POWER_OF_2(size_t n)`

- **作用：**判断 `n` 是否为 2 的幂（如 2、4、8 等）。
- **目的：**伙伴系统中所有内存块大小必须是 2 的幂，该函数用于验证块大小的合法性。

`Get_Order_Of_2(size_t n)`

- **作用：**返回 `n` 对应的幂指数（如 `n=8` 时返回 3，因  $2^3=8$ ）。
- **目的：**将块大小（页数）转换为阶数，用于索引 `free_array` 中的空闲链表。

`Find_The_Small_2(size_t n)`

- **作用：**找到小于等于 `n` 的最大 2 的幂（如 `n=5` 时返回 4）。
- **目的：**初始化内存块时，将连续的 `n` 页划分为最大可能的 2 的幂大小的块。

`Find_The_Big_2(size_t n)`

- **作用：**找到大于等于 `n` 的最小 2 的幂（如 `n=5` 时返回 8）。
- **目的：**分配内存时，根据请求的页数确定最小的满足需求的块大小（必须为 2 的幂）。

## 2. 初始化函数

`buddy_system_init(void)`

- **作用：**初始化伙伴系统，初始化 `free_array` 中所有链表（置为空），设置 `max_order` 和 `nr_free` 为 0。
- **目的：**为伙伴系统运行准备基础数据结构。

`buddy_system_init_memmap(struct Page *base, size_t n)`

- **作用：**将一段连续的 `n` 页物理内存初始化为伙伴系统的空闲块。
- **过程：**
  1. 计算 `n` 对应的最大 2 的幂大小（通过 `Find_The_Small_2`），确定初始块的阶数；
  2. 初始化该块内所有页的属性（清除标志、重置引用计数）；
  3. 将块的起始页加入 `free_array` 中对应阶数的链表，并更新 `max_order` 和 `nr_free`。
- **目的：**将物理内存纳入伙伴系统的管理范围。

## 3. 块分裂函数

`buddy_system_split(size_t n)`

- **作用：**将 `free_array[n]` 链表中的一个大小为  $2^n$  页的块分裂为两个大小为  $2^{(n-1)}$  页的块。
- **过程：**
  1. 从 `free_array[n]` 中取出第一个空闲块，拆分为两个等大的子块（`page1` 和 `page2`）；
  2. 更新子块的阶数属性（设为 `n-1`），并标记为空闲；
  3. 将两个子块加入 `free_array[n-1]` 链表，同时从 `free_array[n]` 中移除原块。
- **目的：**当没有合适大小的块分配时，通过分裂高阶块生成低阶块，满足分配需求。

## 4. 分配函数

`buddy_system_alloc_pages(size_t requested_pages)`

- **作用：**分配 `requested_pages` 个连续物理页，返回起始页指针。
- **过程：**
  1. 检查请求合法性（请求页数  $> 0$  且不超过空闲页总数）；
  2. 计算满足需求的最小 2 的幂块大小（通过 `Find_The_Big_2`），确定目标阶数 `order_of_2`；
  3. 从 `free_array[order_of_2]` 开始查找空闲块，若为空则分裂更高阶的块（通过 `buddy_system_split`），直到找到合适块；
  4. 从链表中移除该块，更新 `nr_free`，返回起始页。
- **目的：**按照伙伴系统规则分配满足需求的内存块。

## 5. 释放与合并函数

`get_buddy(struct Page *page)`

- **作用：**计算给定块的“伙伴块”地址（与当前块大小相同、地址相邻的块）。
- **目的：**释放块时，用于检查伙伴块是否空闲，为合并做准备。

**释放逻辑：**

- **作用：**将释放的块归还给伙伴系统，并尝试与伙伴块合并为更大的块。
- **过程：**
  1. 计算释放块的阶数，将其加入 `free_array` 对应链表；
  2. 检查其伙伴块是否空闲且同阶，若满足则合并为一个高阶块，更新链表；
  3. 重复合并过程，直到无法合并（无空闲伙伴块或达到最大阶数）；
  4. 更新 `nr_free`。
- **目的：**避免内存碎片，通过合并维持大尺寸空闲块。

## 6. 调试函数

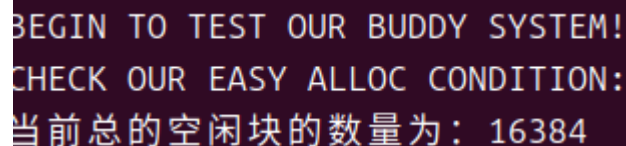
`show_buddy_array(int left, int right)`

- **作用：**显示 `free_array` 中从阶数 `left` 到 `right` 的所有空闲块信息（页数、地址）。
- **目的：**调试时观察空闲块的分布和变化，验证分裂、合并、分配、释放的正确性。

## (四)、正确性验证（基于检测函数）

### 1. `buddy_system_check_easy_alloc_and_free_condition()`

- **测试目标：**验证基本的分配与释放流程。
- **测试内容：**
  - 分配一个中等大小的块（如 4 页），检查 `nr_free` 减少且返回的块地址有效；
  - 释放该块，检查 `nr_free` 恢复且块被正确加入对应空闲链表；
  - 释放后尝试合并伙伴块（若存在），验证合并后块的阶数是否正确。
- **验证点：**基本分配 / 释放逻辑正确，空闲块计数准确。
  - 原始的内存块的布局应该为：16384大小的一个块



```
BEGIN TO TEST OUR BUDDY SYSTEM!  
CHECK OUR EASY ALLOC CONDITION:  
当前总的空闲块的数量为：16384
```

- p0请求10页，程序执行**`buddy_system_alloc_pages`**函数，找到对应的下标为4，但是目前4号位的链表为空，于是往后遍历，遍历到的第一个不为空的链表即为16384的14号链表，于是执行**`split`**操作，将16384大小的内存块分割为下图：

首先 ,p0请求10页

-----当前空闲的链表数组 :-----

No.4的空闲链表有16页 【地址为0xfffffffffc020f598】

No.5的空闲链表有32页 【地址为0xfffffffffc020f818】

No.6的空闲链表有64页 【地址为0xfffffffffc020fd18】

No.7的空闲链表有128页 【地址为0xfffffffffc0210718】

No.8的空闲链表有256页 【地址为0xfffffffffc0211b18】

No.9的空闲链表有512页 【地址为0xfffffffffc0214318】

No.10的空闲链表有1024页 【地址为0xfffffffffc0219318】

No.11的空闲链表有2048页 【地址为0xfffffffffc0223318】

No.12的空闲链表有4096页 【地址为0xfffffffffc0237318】

No.13的空闲链表有8192页 【地址为0xfffffffffc025f318】

-----显示完成 !-----

- 初始的大内存块被分割为了一系列的小块，然后p0取走了我们的16页空闲页
- 按照测试函数，p1再请求10页，相当于取走当前No.4的空闲链表中的16个空页：



然后 ,p1请求10页

-----当前空闲的链表数组:-----

No.5的空闲链表有32页 【地址为0xfffffffffc020f818】

No.6的空闲链表有64页 【地址为0xfffffffffc020fd18】

No.7的空闲链表有128页 【地址为0xfffffffffc0210718】

No.8的空闲链表有256页 【地址为0xfffffffffc0211b18】

No.9的空闲链表有512页 【地址为0xfffffffffc0214318】

No.10的空闲链表有1024页 【地址为0xfffffffffc0219318】

No.11的空闲链表有2048页 【地址为0xfffffffffc0223318】

No.12的空闲链表有4096页 【地址为0xfffffffffc0237318】

No.13的空闲链表有8192页 【地址为0xfffffffffc025f318】

-----显示完成!-----

- 最后p2请求10页，相当于需要将No.5的链表中的空页拆分开来，输出如下结果：



最后 ,p2请求10页

```
-----当前空闲的链表数组:-----  
No.4的空闲链表有16页  【地址为0xfffffffffc020fa98】  
  
No.6的空闲链表有64页  【地址为0xfffffffffc020fd18】  
  
No.7的空闲链表有128页  【地址为0xfffffffffc0210718】  
  
No.8的空闲链表有256页  【地址为0xfffffffffc0211b18】  
  
No.9的空闲链表有512页  【地址为0xfffffffffc0214318】  
  
No.10的空闲链表有1024页  【地址为0xfffffffffc0219318】  
  
No.11的空闲链表有2048页  【地址为0xfffffffffc0223318】  
  
No.12的空闲链表有4096页  【地址为0xfffffffffc0237318】  
  
No.13的空闲链表有8192页  【地址为0xfffffffffc025f318】  
  
-----显示完成!-----
```

- 开始我们的内存收回操作，调用buddy\_system\_free\_pages函数，首先收回p0，也就是要收回16个空页，由于无地址相邻空页表，所以不执行合并，只完成收回。

```
p0的虚拟地址为:0xfffffffffc020f318.
p1的虚拟地址为:0xfffffffffc020f598.
p2的虚拟地址为:0xfffffffffc020f818.
CHECK OUR EASY FREE CONDITION:
释放p0...
Buddy System算法将释放第NO.525127页开始的共16页
释放p0后,总空闲块数目为:16352
-----当前空闲的链表数组:-----
No.4的空闲链表有16页  【地址为0xfffffffffc020f318】
No.4的空闲链表有16页  【地址为0xfffffffffc020fa98】

No.6的空闲链表有64页  【地址为0xfffffffffc020fd18】

No.7的空闲链表有128页  【地址为0xfffffffffc0210718】

No.8的空闲链表有256页  【地址为0xfffffffffc0211b18】

No.9的空闲链表有512页  【地址为0xfffffffffc0214318】

No.10的空闲链表有1024页  【地址为0xfffffffffc0219318】

No.11的空闲链表有2048页  【地址为0xfffffffffc0223318】

No.12的空闲链表有4096页  【地址为0xfffffffffc0237318】

No.13的空闲链表有8192页  【地址为0xfffffffffc025f318】

-----显示完成!-----
```

- 收回p1, 由于地址与上一轮中的0xfffffffffc020f318相邻, 所以完成合并操作, 同时调换链表地址顺序, 保持大小关系。

```
释放p1...
Buddy System算法将释放第NO.525143页开始的共16页
释放p1后,总空闲块数目为:16368
-----当前空闲的链表数组:-----
No.4的空闲链表有16页 【地址为0xfffffffffc020fa98】

No.5的空闲链表有32页 【地址为0xfffffffffc020f318】

No.6的空闲链表有64页 【地址为0xfffffffffc020fd18】

No.7的空闲链表有128页 【地址为0xfffffffffc0210718】

No.8的空闲链表有256页 【地址为0xfffffffffc0211b18】

No.9的空闲链表有512页 【地址为0xfffffffffc0214318】

No.10的空闲链表有1024页 【地址为0xfffffffffc0219318】

No.11的空闲链表有2048页 【地址为0xfffffffffc0223318】

No.12的空闲链表有4096页 【地址为0xfffffffffc0237318】

No.13的空闲链表有8192页 【地址为0xfffffffffc025f318】

-----显示完成!-----
```

- 最后释放p2，地址相邻，我们循环做合并操作，最后还原到原来的内存状态，如下所示。

```
释放p2...
Buddy System算法将释放第NO.525159页开始的共16页
释放p2后,总空闲块数目为:16384
-----当前空闲的链表数组:-----
No.14的空闲链表有16384页 【地址为0xfffffffffc020f318】
-----显示完成!-----
```

## 2. buddy\_system\_check\_difficult\_alloc\_and\_free\_condition()

- **测试目标：**验证复杂场景下的分裂与合并（如混合分配不同大小的块）。
- **测试内容：**
  - 先分配 8 页块，再分配 4 页块（触发 8 页块分裂为两个 4 页块）；
  - 释放 4 页块，检查是否与伙伴块合并为 8 页块；

- 多次交替分配 / 释放不同阶数的块，检查空闲块链表状态是否符合预期。
- **验证点：**多次分裂与合并后，块的阶数、数量、地址均正确，无内存泄漏或重复分配。

结果如图：

```

释放p2...
Buddy System算法将释放第NO.525255页开始的共128页
释放p2后,总空闲块数目为:16384
-----当前空闲的链表数组:-----
No.14的空闲链表有16384页 【地址为0xffffffffc020f318】
-----显示完成!-----

```

### 3. buddy\_system\_check\_min\_alloc\_and\_free\_condition()

- **测试目标：**验证最小块（1 页，阶数 0）的分配与释放。
- **测试内容：**
- 多次分配 1 页块，检查每次分配的块地址不重叠且 `nr_free` 正确减少；
- 释放这些块，检查是否能合并为 2 页、4 页等更大块（验证合并逻辑）。
- **验证点：**最小块的分裂（从高阶块分裂出 1 页块）和合并（多个 1 页块合并为高阶块）正确。

结果如图：

分配p3之后(1页)

-----当前空闲的链表数组:-----

No.0的空闲链表有1页 【地址为0xfffffffffc020f340】

No.1的空闲链表有2页 【地址为0xfffffffffc020f368】

No.2的空闲链表有4页 【地址为0xfffffffffc020f3b8】

No.3的空闲链表有8页 【地址为0xfffffffffc020f458】

No.4的空闲链表有16页 【地址为0xfffffffffc020f598】

No.5的空闲链表有32页 【地址为0xfffffffffc020f818】

No.6的空闲链表有64页 【地址为0xfffffffffc020fd18】

No.7的空闲链表有128页 【地址为0xfffffffffc0210718】

No.8的空闲链表有256页 【地址为0xfffffffffc0211b18】

No.9的空闲链表有512页 【地址为0xfffffffffc0214318】

No.10的空闲链表有1024页 【地址为0xfffffffffc0219318】

No.11的空闲链表有2048页 【地址为0xfffffffffc0223318】

No.12的空闲链表有4096页 【地址为0xfffffffffc0237318】

No.13的空闲链表有8192页 【地址为0xfffffffffc025f318】

Buddy System算法将释放第NO.525127页开始的共1页

-----当前空闲的链表数组:-----

No.14的空闲链表有16384页 【地址为0xfffffffffc020f318】

-----显示完成!-----

#### 4. buddy\_system\_check\_max\_alloc\_and\_free\_condition()

- **测试目标：**验证最大块（ $2^{\text{max\_order}}$  页）的分配与释放。
- **测试内容：**
  - 分配系统支持的最大块，检查是否能直接从 `free_array[max_order]` 中获取；
  - 释放该块，检查是否能正确加入 `free_array[max_order]` 且不被拆分。
  - **验证点：**最大块的分配无需分裂，释放后不合并（无更大阶数），逻辑正确。
  - 直接取走16384个空闲块后的内存情况，程序检测到全链表为空，输出“目前无空闲块!!!”，释放后，内存恢复原状

```
分配p3之后(16384页)
-----当前空闲的链表数组:-----
目前无空闲块!!!
-----显示完成!-----

Buddy System算法将释放第NO.525127页开始的共16384页
-----当前空闲的链表数组:-----
No.14的空闲链表有16384页 【地址为0xffffffffc020f318】
-----显示完成!-----
```

## Challenge2: SLUB 内存分配算法（编程实现）

### (一)、相关文件作用说明

#### 1. slab.h

头文件，定义 SLUB 算法的核心数据结构（`struct slab` 和 `struct slab_cache`），并声明对外提供的接口函数（如缓存创建、对象分配 / 释放）及测试函数。该文件是 SLUB 模块与内核其他部分交互的基础，确保数据结构和函数的一致性。

#### 2. slab.c

核心实现文件，包含 SLUB 算法的完整逻辑。具体包括缓存的创建与销毁、slab 的初始化与管理、对象的分配与释放，以及空闲对象链表的维护。同时依赖内核现有页分配器（如伙伴系统）获取物理页，是 SLUB 算法的功能载体。

#### 3. slab\_test.c

测试文件，提供一系列测试用例验证 SLUB 算法的正确性。通过模拟不同场景下的对象分配、释放、边界条件（如空缓存分配、全满 slab 释放），检查缓存状态、对象地址有效性及计数准确性，确保算法无内存泄漏或逻辑错误。

## (二)、核心结构体说明

### 1. struct slab (管理单个物理页内的对象)

```
struct slab {
    struct list_head list;           // 链表节点，用于将 slab 接入缓存的三类链表
    void *start;                     // slab 对应物理页的起始地址（对象存储的基地址）
    void *free_list;                 // 空闲对象链表头，通过指针链管理空闲对象
    size_t obj_size;                 // 单个对象的大小（与所属缓存的 obj_size 一致）
    int free_cnt;                     // 当前 slab 中剩余的空闲对象数量
    int total_cnt;                   // 当前 slab 中对象的总数量（= PAGE_SIZE / obj_size）
};
```

- **核心作用：**管理一页物理内存内的同大小对象，记录空闲对象分布和数量，通过 `list` 节点关联到缓存的对应链表（满 / 部分满 / 空）。
- **关键字段：**`free_list` 采用“指针链”设计，每个空闲对象的首地址存储下一个空闲对象的地址，实现高效的空闲对象查找与回收。

### 2. struct slab\_cache (管理同一类型对象的所有 slab)

```
struct slab_cache {
    const char *name;                // 缓存名称（如 "kmalloc-64"，用于调试标识）
    size_t obj_size;                 // 该缓存中对象的固定大小（用户指定）
    size_t align;                    // 对象的内存对齐要求（如 8 字节对齐，优化访问效率）
    struct list_head slabs_full;      // 无空闲对象的 slab 链表（分配时跳过）
    struct list_head slabs_partial;   // 有部分空闲对象的 slab 链表（分配优先选择）
    struct list_head slabs_empty;     // 全空闲对象的 slab 链表（无部分满时使用）
};
```

- **核心作用：**作为 SLUB 算法的“管理层”，聚合所有同类型对象的 slab，通过三类链表区分 slab 状态，实现分配时的高效 slab 选择和释放时的状态更新。
- **设计逻辑：**按 slab 空闲状态分类管理，避免遍历所有 slab 查找空闲对象，提升分配效率。

## (三)、核心函数作用与目的

### 1. 缓存管理函数

#### (1) slub\_create\_cache (创建新缓存)

```
struct slab_cache *slub_create_cache(const char *name, size_t obj_size, size_t align);
```

- **作用：**初始化一个新的 slab 缓存，为特定大小的对象分配提供“容器”。
- **核心流程：**
  1. 调用内核现有分配器（如 `kmalloc`）分配 `struct slab_cache` 结构体内存；
  2. 初始化缓存元数据（名称、对象大小、对齐要求）；
  3. 初始化三类 slab 链表（`slabs_full` / `slabs_partial` / `slabs_empty`）为空链表。
- **目的：**为同大小对象的批量分配建立管理单元，避免重复初始化 slab 逻辑。

#### (2) slub\_destroy\_cache (销毁缓存)



```
void slub_destroy_cache(struct slab_cache *cache);
```

- **作用：**释放缓存占用的所有资源，包括所有 slab 对应的物理页和结构体内存。
- **核心流程：**
  1. 遍历缓存的三类 slab 链表，逐个释放 slab 对应的物理页（`pmm_free_pages`）和 slab 结构体（`kfree`）；
  2. 释放 `struct slab_cache` 结构体本身。
- **目的：**在缓存不再使用时回收资源，避免内存泄漏。

## 2. 对象分配函数

`slub_alloc`（从缓存分配一个对象）

```
void *slub_alloc(struct slab_cache *cache);
```

- **作用：**从指定缓存中分配一个初始化后的对象，优先选择高效路径（部分满 slab）。
- **核心流程：**
  1. **选择 slab：**优先从 `slabs_partial` 取 slab；无则从 `slabs_empty` 取并移至 `slabs_partial`；均无则新分配物理页创建 slab；
  2. **分配对象：**从 slab 的 `free_list` 取出首个空闲对象，更新 `free_list` 指向 next 空闲对象，`free_cnt` 减 1；
  3. **更新 slab 状态：**若 `free_cnt` 变为 0，将 slab 从 `slabs_partial` 移至 `slabs_full`；
  4. **初始化对象：**用 `memset` 清零对象内存（避免脏数据），返回对象地址。
- **目的：**实现高效的小对象分配，避免直接调用页分配器产生内存碎片（一页可存储多个小对象）。

## 3. 对象释放函数

`slub_free`（释放对象到缓存）

```
void slub_free(struct slab_cache *cache, void *obj);
```

- **作用：**将对象回收到所属缓存的对应 slab，更新空闲链表和 slab 状态，支持后续复用。
- **核心流程：**
  1. **查找 slab：**遍历缓存的三类链表，判断对象地址是否在某个 slab 的 `start` 到 `start + PAGE_SIZE` 范围内，定位所属 slab；
  2. **回收对象：**将对象接入 slab 的 `free_list` 头部（`*(void**)obj = free_list; free_list = obj`），`free_cnt` 加 1；
  3. **更新 slab 状态：**
    - 若 `free_cnt` 从 0 变为 1（slab 从满变部分满），将 slab 从 `slabs_full` 移至 `slabs_partial`；
    - 若 `free_cnt` 等于 `total_cnt`（slab 全空），将 slab 从 `slabs_partial` 移至 `slabs_empty`。
- **目的：**回收对象并维护缓存状态，实现对象复用，减少新页分配频率。



## (四)、测试验证

### 测试验证函数

通过单流程测试验证核心功能：

1. 创建一个名为 `test-cache-64` 的缓存，指定对象大小 64 字节、8 字节对齐；
2. 从缓存中分配 3 个对象，写入字符串数据并验证数据可正常访问；
3. 释放这 3 个对象，再次分配 1 个对象，验证释放的内存可被复用；
4. 销毁缓存，确认无资源泄漏（通过后续编译运行无崩溃验证）。

```
#include "slub.h"
#include <stdio.h>
#include <string.h>
#include <assert.h>

// 基础功能测试函数
void slub_basic_test() {
    printf("=== Starting SLUB Basic Function Test ===\n");

    // 1. 创建缓存（对象大小64字节，8字节对齐）
    struct slab_cache *cache = slub_create_cache("test-cache-64", 64, 8);
    assert(cache != NULL && "Failed to create cache");
    printf("Cache created: name=%s, obj_size=%zu, align=%zu\n",
        cache->name, cache->obj_size, cache->align);

    // 2. 分配3个对象并写入测试数据
    void *obj1 = slub_alloc(cache);
    void *obj2 = slub_alloc(cache);
    void *obj3 = slub_alloc(cache);
    assert(obj1 != NULL && "Failed to alloc obj1");
    assert(obj2 != NULL && "Failed to alloc obj2");
    assert(obj3 != NULL && "Failed to alloc obj3");
    printf("Allocated 3 objects: %p, %p, %p\n", obj1, obj2, obj3);

    // 写入数据验证内存可访问性
    strcpy((char*)obj1, "SLUB Test Object 1");
    strcpy((char*)obj2, "SLUB Test Object 2");
    strcpy((char*)obj3, "SLUB Test Object 3");
    printf("Object 1 data: %s\n", (char*)obj1);
    printf("Object 2 data: %s\n", (char*)obj2);
    printf("Object 3 data: %s\n", (char*)obj3);

    // 3. 释放对象
    slub_free(cache, obj1);
    slub_free(cache, obj2);
    slub_free(cache, obj3);
    printf("Freed 3 objects\n");

    // 4. 再次分配对象，验证释放的对象可复用（地址可能与之前相同）
    void *obj4 = slub_alloc(cache);
    assert(obj4 != NULL && "Failed to alloc obj4 (reuse)");
```

```
printf("Re-allocated object: %p (should reuse freed memory)\n", obj4);

// 5. 销毁缓存
slub_destroy_cache(cache);
printf("Cache destroyed\n");

printf("=== SLUB Basic Function Test Passed ===\n\n");
}

// 测试入口
int main() {
    slub_basic_test();
    return 0;
}
```

## Challenge3: 硬件的可用物理内存范围的获取方法（思考题）

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

1、**通过固件获取内存信息**：固件会提前完成初步硬件探测并以标准化格式提供给 OS，其中 BIOS 在启动时生成 SMBIOS 表格，UEFI 生成 UEFI 内存映射表，OS 启动后可通过 BIOS 的 0x15 中断或 UEFI 系统调用，直接读取这些表格中的物理内存地址范围及可用、已占用、保留等类型信息；同时，对于 PCIe 等总线设备，OS 还能通过访问其配置空间（如 PCIe 配置空间的 BAR 寄存器）获取设备占用的物理内存地址，进而间接排除已被硬件占用的区域，确定剩余可用内存范围。

2、**执行主动内存探测算法**：当固件信息不可靠或不存在（如极简嵌入式系统）时，OS 会执行主动内存探测算法，通过主动写入和读取数据的方式逐个地址验证内存是否可用，其基础探测逻辑是从预设起始地址（如 0x00000000）开始，向目标内存地址写入特定测试值（如 0xDEADBEEF）后立即读取该地址的值，若读取结果与写入值一致，则说明该地址对应的物理内存存在且可用；完成当前地址探测后，OS 会按字节或按页对齐逐步递增地址，直到遇到连续多次读取失败的地址，以此确定物理内存的上限，同时还会标记出因硬件占用（如显存、IO 映射）导致读取失败的“空洞”区域，避免 OS 使用

3、**结合硬件规格与默认规则兜底**：在极端场景下（如无固件的裸机系统），OS 会依赖硬件手册的固定规格和行业默认规则确定初始可用内存范围，其中部分硬件会将低地址区域（如 x86 架构的前 1MB 内存）默认作为可用内存，OS 可先使用该区域完成初始化，再通过后续探测扩展内存范围，而嵌入式芯片（如 ARM Cortex-M 系列）的手册会明确标注物理内存的起始地址和大小（如 0x20000000 - 0x2000FFFF，共 64KB RAM），OS 可直接根据手册参数定义内存范围，无需额外探测。