Lab2 实验报告

小组成员：唐显达 2210878　王禹衡 2213040　苑译元 2210983

练习1：理解first-fit 连续物理内存分配算法

答：

1. first-fit算法的整体思路：

first-fit算法是操作系统用来**管理内存分配的算法**，主要通过**维护一个链表的数据结构进行内存的分配。**在这个算法中，内存分配器会维护一个空闲块的链表，当需要分配内存为**n**的内存块的时候，分配器会遍历这个链表，找到**第一个满足要求（容量大小可以满足请求）**的内存块，将它分配给对应的请求。如果所找到的块比请求的内存大，则会将其拆分，剩余的部分仍然作为空闲块加入到空闲链表中。

2. 分析`default_init`，`default_init_memmap`，`default_alloc_pages`，`default_free_pages`等相关函数：

- `default_init`:初始化free_list链表，用于存储所有的空闲内存块，并将nr_free（记录空闲页面数量）设为0。这一步是**初始化物理内存分配器的第一步**，确保空闲链表为空，并开始跟踪空闲页面数量。
- `default_init_memmap`:初始化一个空闲的内存块，整体的思路为：
  - 首先遍历每个页面，**初始化标志位和页面的属性，而后将页面的引用计数设置为0**;
  - 而后将这段内存块的**第一个页面的`property`字段设为块的总页面数**，并将其**标记为`PG_property`**;
  - 将这段内存块插入空闲链表，并更新总的空闲页面的数量nr_free。
- `default_alloc_pages`：用于**分配给定大小的内存块。**如果剩余空闲内存块大小多于所需的内存区块大小，则从链表中查找大小超过所需大小的页，并更新该页剩余的大小。
  - 首先遍历空闲块的列表，查找到第一个可以保证块的大小大于等于n的空闲块
  - 如果找到了合适的块，那么会将块分割成两部分，一部分用于分配，一部分保留在列表中，如果该块的大小大于`n`，则将剩余部分重新加入空闲链表。
  - 最后，调整其`property`属性。将其标记为已经分配的情况，修改`nr_free`的数量。
- `default_free_pages`：用于**释放内存块。**将释放的内存块按照顺序插入到空闲内存块的链表中，并合并与之相邻且连续的空闲内存块。
  - 首先将页面的属性重置，将对应的计数位恢复位0；
  - 而后将对应的释放的页面添加到当前的空闲内存块链表中，而后尝试合并空闲块
  - 分别分析释放的块与相邻的前后两个页面是否可以合并，如果满足合并的条件，那么将他们合并为一个更大的空闲块，否则不合并。
  - 最后，更新对应的块的计数。

3. first-fit算法的优化分析：

- **提高碎片内存资源的利用率：**由于first-fit算法会**直接选择在查找过程中的第一个满足条件的内存块**，这样可能会导致分配算法将原来大的内存块切分为小的内存块，这会**导致大量的碎片化内存**，造成内存碎片化严重，影响内存的分配和使用。

- **优化分配的策略**：引入更加高效合理的分配策略，例如练习2中提到的使用best-fit算法或者buddy_system分配方法，**选择最合适或紧邻上一次分配位置的块进行分配**，减少碎片的生成。
- **优化空闲块的搜索算法**：目前查询空闲块的算法是通过遍历的方法进行查找的，这样算法的性能不是很好，可以**尝试构建树型搜索等相关的搜索算法**实现对于空闲块的搜索和匹配。

练习2：实现 Best-Fit 连续物理内存分配算法

与练习一中提到的first-fit算法相比，Best-Fit算法在每次查找相关能够满足要求的最小的空闲块进行分配。下面我们对设计的思路进行分析：

设计实现过程：

```
// 遍历空闲链表，查找满足需求的空闲页框
// 如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量
while ((le = list_next(le)) != &free_list)
    {
        struct Page* p = le2page(le, page_link);
        if (p->property >= n && p->property<min_size)
        {
            page = p;
            min_size=p->property;
        }
    }
```

如上的代码所示，我们只需要在first-fit的基础上修改 `best_fit_alloc_pages` 函数即可完成内存分配算法的修改。我么在每次进行顺序查找的时候，添加一个比较满足条件的空闲块大小的功能，最终选择满足条件的最小的空闲块，就可以实现对应的算法。

扩展练习Challenge：buddy system（伙伴系统）分配算法

buddy system特点分析：

根据前面练习1，练习2中的思路，无论采用first-fit算法还是best-fit算法，我们都无法完全地利用整个的内存空间，都会造成部分内存空间的碎片化，浪费掉的内存是很大的，因此扩展练习的要求，我们需要实现buddy system的分配算法，从而实现更加高效的内存分配算法。

根据实验指导书中，关于二叉树结构的buddy system的介绍，我们总结出buddy system的以下特点：

伙伴系统分配算法就是将内存按2的幂进行划分，相当于分离出若干个块大小一致的空闲链表，搜索该链表并给出同需求最佳匹配的大小。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，因为**按2的幂划分块**，如果碰上66单位大小，那么必须划分128单位大小的块。

下面进行算法的设计思路分析：

1. 初始化整个伙伴系统

```
 //初始化指定阶层的空闲内存块链表
static void init_free_area(int order) {
    list_init(&(free_area[order].free_list));  // 初始化链表为空
    free_area[order].nr_free = 0;  // 空闲块数量初始化为0
}
//初始化所有阶层的空闲内存块链表
static void buddy_system_init(void) {
    for (int i = 0; i < MAX_ORDER; i++) {
        init_free_area(i);
    }
}
```

和之前的算法类似，我们实现对链表的初始化。**初始化空闲链表为空**，而后将对应**的nr_free设置为0，代表空闲块的数量为0；**

同时，我们编写函数`buddy_system_init`函数，从而初始化所有阶层的链表为空闲状态，在不同的阶层中，调用`init_free_area`。

2. 初始化物理映射

在完成了初始化伙伴系统之后，我们将**对应的物理内存地址映射到伙伴系统**，将指定范围内的物理内存页块加入到伙伴系统的空闲列表中，便于后续伙伴系统的实现。

```
static void buddy_system_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);  // 确保有页可用
    struct Page *p = base;
    // 遍历每个页，清除属性和引用计数，确保页被释放
    for (; p != base + n; p++) {
        assert(PageReserved(p));  // 页必须被保留
        p->flags = p->property = 0;  // 清除页的标志和属性
        set_page_ref(p, 0);  // 设置引用计数为0}
    size_t offset = 0;
    while (n > 0) {
        uint32_t order = 0;
        while ((1 << order) <= n && order < MAX_ORDER) {
            order += 1;}
        if (order > 0) {
            order -= 1;}
        p = base + offset;
        p->property = 1 << order;
        SetPageProperty(p);
        list_add(&(free_list(order)), &(p->page_link));
        nr_free(order) += 1;
        offset += (1 << order);
        n -= (1 << order);
    }
}
```

整体代码的执行顺序：

1. **初始化并检查输入参数**：在算法的执行过程中，首先将对应页的标志位和属性都恢复为初始状态，而后设置对应的引用计数；
2. **在伙伴系统中寻找合适的块**：在寻找合适的块的时候，首先要确定
3. **设置页块的属性**：将其加入空闲链表；
4. **更新偏移量和剩余页数**：继续处理下一个页块，直到所有页处理完毕。

3. 内存块分割

在我们的内存分配算法执行的过程当中，可能会由于内存块的大小，与我们需求的内存块的大小相差较大，此时，我们可以**将原来的相对较大的内存块分割为两个相对较小的内存块**，从而防止对内存的浪费：

```
static void split_page(int order) {
    assert(order > 0 && order < MAX_ORDER);
    if (list_empty(&(free_list(order)))) {
        split_page(order + 1);}
    if (list_empty(&(free_list(order)))) {
        return;}
    list_entry_t *le = list_next(&(free_list(order)));
    struct Page *page = le2page(le, page_link);
    list_del(&(page->page_link));
    nr_free(order) -= 1;
    uint32_t size = 1 << (order - 1);
    struct Page *buddy = page + size;
    page->property = size;
    buddy->property = size;
    SetPageProperty(page);
    SetPageProperty(buddy);
    list_add(&(free_list(order - 1)), &(page->page_link));
    list_add(&(free_list(order - 1)), &(buddy->page_link));
    nr_free(order - 1) += 2;
}
```

> 代码在完成了对输入的参数的检查之后，会递归地检查当前级别的空闲页块链表，如果`(list_empty(&(free_list(order))))`条件仍然满足，那么代表链表仍然为空，则没有可用的页块，直接返回，函数终止。否则，从当前的order级别的链表中提取一个空闲的也快，而后**计算将其拆分后每个子页块的大小，计算出对应的页块的位置，最后重新将对应的页块加入更低一级别的空闲链表中，调整其对应的参数和相关的标志位。**

4. 内存块合并

add_page函数实现了将内存块加入了对应的阶层的空闲链表中，我们直接调用已经封装好的对应的链表的函数。

```
// 将内存块插入到指定阶层的空闲链表中
static void add_page(uint32_t order, struct Page *base) {
    list_add(&(free_list(order)), &(base->page_link));
}
static void merge_page(uint32_t order, struct Page *base) {
```

```
    if (order >= MAX_ORDER - 1) {
        // 当达到最高阶层时，直接将页面添加回空闲链表
        add_page(order, base);
        nr_free(order) += 1;
        return;}
    size_t size = 1 << order;
    uintptr_t addr = page2pa(base);
    uintptr_t buddy_addr = addr ^ (size << PGSHIFT);
    struct Page *buddy = pa2page(buddy_addr);
    if (buddy->property != size || !PageProperty(buddy)) {
        // 伙伴不可合并
        add_page(order, base);
        nr_free(order) += 1;
        return;}
    // 从当前阶层中移除伙伴
    list_del(&(buddy->page_link));
    ClearPageProperty(buddy);
    nr_free(order) -= 1;
    // 选择地址较小的块作为新的基地址
    if (buddy < base) {base = buddy;}
    // 设置新的属性值
    base->property = size << 1;
    ClearPageProperty(base);
    // 递归合并到更高阶层
    merge_page(order + 1, base);
}
```

> 首先会检查当前的链表是否已经达到了最大的阶层，如果已经到达了最大的阶层，那么会直接将新的页块加入到空闲链表的最后的位置上。
>
> 如果没有到达最大的内存的阶层，那么会获得对应伙伴页块的地址，找到伙伴页块，而后尝试，是否可以和伙伴页块进行合并。
>
> 如果**不可以合并，那么将页块加入空闲的链表中，如果可以合并，那么直接移除伙伴页块并更新空闲链表，选择对应的合并后的基地址**，更新合并后的属性，记录对应阶层的空闲块的数量和属性，完成合并操作。

5. 分配内存块

```
static struct Page *buddy_system_alloc_pages(size_t n) {
    assert(n > 0);  // 确保要分配的页数大于0
    if (n > (1 << (MAX_ORDER - 1))) {  // 请求的页数超过最大块大小
        return NULL;  // 无法满足请求
    }
    struct Page *page = NULL;
    // 正确计算需要的最小阶层
    uint32_t order = 0;
    while ((1 << order) < n && order < MAX_ORDER) {
        order += 1;
    }
    if (order >= MAX_ORDER) {
        return NULL;
    }
    uint32_t current_order = order;
    // 查找可用的块，如果当前阶层没有，则向上寻找
```

```
    while (current_order < MAX_ORDER && list_empty(&(free_list(current_order))))
{
        current_order += 1;
    }
    if (current_order == MAX_ORDER) {
        return NULL;
    }
    // 逐级拆分，直到达到所需的阶层
    while (current_order > order) {
        split_page(current_order);
        current_order -= 1;
    }
    // 从空闲链表中获取块
    list_entry_t *le = list_next(&(free_list(order)));
    page = le2page(le, page_link);
    list_del(&(page->page_link));
    nr_free(order) -= 1;
    ClearPageProperty(page);
    return page;
}
```

分配块的内存需要使用到前面已经提到的`split_page`函数，来将较大的内存块分割为较小的内存块。

整体的函数思路为：**首先判断传入函数中的需要分配的内存的大小**，若需要分配的页的大小已经超过了我们规定限制的最大的限度，那么会返回空值，如果符合我们进行分配的要求，那么需要根据当前页的大小，开始寻找可以满足要求的对应的阶层的页，**判断当前阶层有没有可以使用的块**，如果当前的阶层没有可以使用的页，那么继续向上搜索，如果找到了对应的阶层，先判断当前的页是否属于当前的阶层，如果当前的页无法完全使用分配的页，则调用`split_page`函数，将对应的页块分割，并整理对应的基层，**最后从空闲列表中获得符合大小的块。完成依次内存的分配。**

6. 回收内存块

```
// 释放页面，将其重新加入空闲链表中
static void buddy_system_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(IS_POWER_OF_2(n));  // 页数必须是2的幂
    assert(n <= (1 << (MAX_ORDER - 1)));  // 页数不能超过最大阶层的块大小
    struct Page *p = base;
    // 遍历每个页，清除标志位和引用计数
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);}
    base->property = n;  // 设置基础块大小
    SetPageProperty(base);
    uint32_t order = 0;  // 计算对应的阶层
    size_t temp = n;
    while (temp > 1) {
        temp >>= 1;
        order++;}
    // 尝试合并
    merge_page(order, base);
}
```

除了需要分配内存以外，我们还需要进行回收内存对应的任务，这部分首先会确保页的数量有效，保证页的数量为2的幂，如果不是2的幂，那么处于对空闲链表的保护，我们不会将页进行回收。

而后我们进行对标志位和属性的清除，体现出我们已经将对应的页清除了，最后调用`merge_page`函数将对应的页表合并。

7. 伙伴系统测试程序

> 针对我们的伙伴系统，我们实现了测试程序。经过验证我们的buddy system算法是有效的。

```
// 检查伙伴系统是否正常运行
static void buddy_system_check(void) {
    size_t total_free_pages = buddy_system_nr_free_pages();
    cprintf("开始伙伴系统检查：total_free_pages = %d\n", total_free_pages);
    // 分配一个页面
    struct Page *p0 = buddy_system_alloc_pages(1);
    assert(p0 != NULL);
    cprintf("分配了1个页面。\n");
    // 检查空闲页面数量是否减少了1
    size_t free_pages_after_alloc1 = buddy_system_nr_free_pages();
    assert(free_pages_after_alloc1 == total_free_pages - 1);
    // 分配另一个页面
    struct Page *p1 = buddy_system_alloc_pages(1);
    assert(p1 != NULL);
    assert(p0 != p1);  // 确保分配了不同的页面
    cprintf("又分配了1个页面。\n");
    // 检查空闲页面数量是否又减少了1
    size_t free_pages_after_alloc2 = buddy_system_nr_free_pages();
    assert(free_pages_after_alloc2 == free_pages_after_alloc1 - 1);
    // 释放第一个页面
    buddy_system_free_pages(p0, 1);
    cprintf("释放了第一个分配的页面。\n");
    // 检查空闲页面数量是否增加了1
    size_t free_pages_after_free1 = buddy_system_nr_free_pages();
    assert(free_pages_after_free1 == free_pages_after_alloc2 + 1);
    // 释放第二个页面
    buddy_system_free_pages(p1, 1);
    cprintf("释放了第二个分配的页面。\n");
    // 检查空闲页面数量是否又增加了1，应当恢复到初始值
    size_t free_pages_after_free2 = buddy_system_nr_free_pages();
    assert(free_pages_after_free2 == total_free_pages);
    cprintf("单个页面的分配和释放测试通过。\n");
    // 现在尝试分配一个页面块
    size_t n = 4; // 尝试分配4个页面
    struct Page *p2 = buddy_system_alloc_pages(n);
    assert(p2 != NULL);
    cprintf("分配了%d页面。\n", n);
    // 检查空闲页面数量是否减少了n
    size_t free_pages_after_alloc_n = buddy_system_nr_free_pages();
    assert(free_pages_after_alloc_n == free_pages_after_free2 - n);
    // 释放页面块
    buddy_system_free_pages(p2, n);
    cprintf("释放了%d个页面。\n", n);
    // 检查空闲页面数量是否增加了n，应当恢复到初始值
```

```
    size_t free_pages_after_free_n = buddy_system_nr_free_pages();
    assert(free_pages_after_free_n == total_free_pages);
    cprintf("分配和释放%d个页面的测试通过。\n", n);
    // 尝试分配最大可能的块
    n = 1 << (MAX_ORDER - 1); // 最大块大小
    struct Page *p3 = buddy_system_alloc_pages(n);
    assert(p3 != NULL);
    cprintf("分配了最大块大小%d个页面。\n", n);
    // 检查空闲页面数量是否减少了n
    size_t free_pages_after_alloc_max = buddy_system_nr_free_pages();
    assert(free_pages_after_alloc_max == total_free_pages - n);
    // 尝试再分配一个页面，应当成功，因为还有剩余的空闲页面
    struct Page *p4 = buddy_system_alloc_pages(n);
    assert(p4 == NULL);
    cprintf("无法生成两个相同的最大块\n,因此，通过了最大页的测试。\n");
    // 释放最大块
    buddy_system_free_pages(p3, n);
    cprintf("释放了大小%d个页面。\n", n);
    // 释放额外的页面
    //buddy_system_free_pages(p4, n);
    //cprintf("释放了额外的页面。\n");
    // 检查空闲页面数量是否恢复到初始值
    size_t free_pages_after_free_max = buddy_system_nr_free_pages();
    assert(free_pages_after_free_max == total_free_pages);
    cprintf("最大块的分配和释放测试通过。\n");
    cprintf("所有伙伴系统检查均通过。\n");
}
```

在pmm.c模块中，我们更换采用的页面管理方法的指针，进行测试，通过make qemu指令，我们成功的实现了对应的分配算法。

扩展练习Challenge：任意大小的内存单元slub分配算法（需要编程）

通过实验指导书和给出的链接可以知道，在Linux内核当中，SLUB（Slab Allocator）的分配算法可以高效管理内存的算法。但它的实际实现非常复杂，需要考虑缓存对齐、NUMA 等多种系统性能优化问题，目前我们的 uCore 作为一个实验性质的操作系统，仅仅实现了一种结合 SLAB 和 SLUB 特性的分配器，带有slub思想的算法。

1、简化版 SLUB 算法概述

1. **数据结构简化：**
   将 Linux 内核中的 `kmem_cache`、`kmem_cache_cpu` 和 `kmem_cache_node` 三个结构体合并为一个 `kmem_cache_t` 结构体。该结构体用于管理存储对象的内存块，维护三个链表：
   - **slabs_full**：已分配满的 slab
   - **slabs_partial**：部分被分配的 slab
   - **slabs_free**：未分配的 slab
2. **内存管理方式：**
   - **固定 Slab 大小：** 每个 Slab 大小为一页（通常为 4096 字节）。

- **对象大小限制：** 对象大小限定为 16、32、64、128、256、512、1024 和 2048 字节，不允许创建大于一页的对象。

3. **静态链表管理空闲块：**
   复用 Page 数据结构，将 Slab 元数据保存在 Page 结构体中，并通过静态链表管理空闲块。

4. **内存释放方式：**
   SLUB 算法中的自动回收机制被移除，改由程序员手动管理 Slab 的释放。

2、核心代码分析

1. 数据结构定义

```
struct kmem_cache_t {
    list_entry_t slabs_full;
    list_entry_t slabs_partial;
    list_entry_t slabs_free;
    uint16_t objsize;
    uint16_t num; // 每个 Slab 保存的对象数量
    void (*ctor)(void *, struct kmem_cache_t *, size_t); // 构造函数
    void (*dtor)(void *, struct kmem_cache_t *, size_t); // 析构函数
    char name[CACHE_NAMELEN]; // 仓库名称
    list_entry_t cache_link;
};

struct slab_t {
    int ref; // 页的引用次数
    struct kmem_cache_t *cachep; // 指向仓库对象的指针
    uint16_t inuse; // 已分配的对象数量
    int16_t free; // 下一个空闲对象的偏移量
    list_entry_t slab_link;
};
```

2. 分配器初始化：`kmem_init()`

```
void kmem_init() {
    cache_cache.objsize = sizeof(struct kmem_cache_t);
    cache_cache.num = PGSIZE / (sizeof(int16_t) + sizeof(struct kmem_cache_t));
    list_init(&(cache_cache.slabs_full));
    list_init(&(cache_cache.slabs_partial));
    list_init(&(cache_cache.slabs_free));
    list_add(&(cache_chain), &(cache_cache.cache_link));

    for (int i = 0, size = 16; i < SIZED_CACHE_NUM; i++, size *= 2) {
        sized_caches[i] = kmem_cache_create(sized_cache_name, size, NULL, NULL);
    }
    check_kmem(); // 测试分配器
}
```

3. 创建仓库：`kmem_cache_create()`

```
struct kmem_cache_t *kmem_cache_create(const char *name, size_t size,
    void (*ctor)(void *, struct kmem_cache_t *, size_t),
    void (*dtor)(void *, struct kmem_cache_t *, size_t)) {

    struct kmem_cache_t *cachep = kmem_cache_alloc(&cache_cache);
    cachep->objsize = size;
    cachep->num = PGSIZE / (sizeof(int16_t) + size);
    memcpy(cachep->name, name, CACHE_NAMELEN);
    list_init(&(cachep->slabs_full));
    list_init(&(cachep->slabs_partial));
    list_init(&(cachep->slabs_free));
    list_add(&(cache_chain), &(cachep->cache_link));
    return cachep;
}
```

4. 分配对象：`kmem_cache_alloc()`

```
void *kmem_cache_alloc(struct kmem_cache_t *cachep) {
    list_entry_t *le = !list_empty(&(cachep->slabs_partial))
                       ? list_next(&(cachep->slabs_partial))
                       : list_next(&(cachep->slabs_free));

    struct slab_t *slab = le2slab(le, page_link);
    void *kva = slab2kva(slab);
    int16_t *bufctl = kva;
    void *objp = bufctl + slab->free * cachep->objsize;

    slab->inuse++;
    slab->free = bufctl[slab->free];

    if (slab->inuse == cachep->num)
        list_add(&(cachep->slabs_full), le);
    else
        list_add(&(cachep->slabs_partial), le);

    return objp;
}
```

3、测试样例

测试样例通过初始化仓库、分配对象、释放对象和验证链表状态来确保分配器的功能正确。

```
static void check_kmem() {
    struct kmem_cache_t *cp0 = kmem_cache_create("test_cache", 2046, NULL, NULL);
    struct test_object *p0, *p1, *p2, *p3, *p4, *p5;

    p0 = kmem_cache_alloc(cp0);
    p1 = kmem_cache_alloc(cp0);
    p2 = kmem_cache_alloc(cp0);
    kmem_cache_free(cp0, p2);
```

```
    assert(list_length(&(cp0->slabs_partial)) == 1);
    assert(list_length(&(cp0->slabs_full)) == 2);

    kmem_cache_destroy(cp0);
    cprintf("check_kmem() succeeded!\n");
}
```

```
free memory: [0x0000000080348000, 0x0000000087ffffff]
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
check_kmem() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
```

借鉴了一些测试样例和代码，从结果来看，check_kmem(),成功了.

扩展练习Challenge：硬件的可用物理内存范围的获取方法

1. 通过BIOS或固件获取内存范围：

> 硬件抽象层（HAL）和BIOS/UEFI固件的关系：
>
> - HAL位于操作系统内部，提供操作系统与硬件之间的抽象层，用于编写操作系统代码，使其与硬件无关。
> - BIOS/UEFI固件是计算机启动时的底层软件，负责硬件初始化和引导操作系统。它们包含了关于硬件和系统配置的信息，包括可用物理内存范围。
> - HAL可以与不同的操作系统配合使用，而BIOS/UEFI固件通常是特定于计算机硬件的。
> - HAL的目标是提供一个硬件抽象层，使操作系统能够在多个硬件平台上运行，而BIOS/UEFI的目标是引导和初始化计算机硬件，为操作系统的加载和执行提供环境。

- 常会通过**内存映射表**（Memory Map）向操作系统提供物理内存的详细信息。
- **方法**：操作系统可以在启动时通过调用BIOS/UEFI提供的接口（如`e820`内存映射）来查询可用的物理内存区域。
    - 操作系统可以读取该映射，了解哪些物理内存区域是可用的，哪些是保留的，从而构建内存管理系统。

2. 通过ACPI表获取内存信息：

ACPI（Advanced Configuration and Power Interface）是一种标准，用于描述计算机硬件配置和电源管理。操作系统可以使用ACPI表格来获取有关系统硬件的信息，包括内存范围。

- **方法**：操作系统可以读取ACPI提供的内存相关表（例如`SRAT`表，System Resource Affinity Table），其中包含内存的拓扑信息，包括哪些区域是可用的，哪些区域是为设备保留的。
- 通过分析ACPI表，操作系统可以获得整个系统的内存映射，从而有效管理可用内存。

3. 操作系统启动参数：

操作系统在启动时通常可以接收一些参数，这些参数可以包含关于物理内存范围的信息。这些参数通常由引导加载程序传递给操作系统，可以分段检测物理内存是否被占用，获得未被占用的物理内存。

4. 内存映射和扫描：

操作系统可以通过内存映射和扫描来查找可用的物理内存范围(这通常需要在较低级别的硬件访问和内存管理代码中进行)，通过扫描内存控制器或硬件相关的寄存器，操作系统可以确定哪些内存范围是可用的。

实验总结：

本次实验主要实现了操作系统内核对于物理内存的分配和管理工作，我们主要的工作是完成对于页面分配算法的设计。

根据实验指导书的介绍，操作系统处理内存分配的算法有很多种：

- **First Fit 算法**：当有申请内存的请求的时候，这种算法的思路是会先遍历空闲页链表，找到其中第一个大小满足要求的空闲页块，而后进行分配；释放的时候会直接将对应的页块加入链表的尾部，并在必要的时候合并页块，以减少碎片化的内存；
- **Best Fit算法**：整体的思路与First Fit算法类似，只是在进行内存块的筛选的时候，从First Fit算法中的找到第一个满足要求的内存块就结束分配，修改为遍历整个链表，找到不仅满足内存申请的需求，同时是所有满足要求的块中，最小的那个块。这种方法，相比First Fit算法相比，对于内存的利用率更高，减少了碎片化的内存；
- **buddy system伙伴系统分配算法**：伙伴系统分配算法主要用于对内存页块的管理，有链式和树状两种类型，主要是由于前两种的链式内存管理算法无法很好的利用碎片化的内存，因此我们选择使用伙伴系统算法进行内存块的分配；
- **slub算法**：是一种内核分配器，主要用于优化内核的小对象分配。它是 Linux 内核的内存分配子系统的一部分，用于取代 SLOB（Simple List of Blocks）和 SLAB 分配器，它通过缓存（Cache）和分块（Slab）结构管理内存，将同一大小的对象放在一个缓存中，并使用部分分块（Partial Slabs）策略来回收和分配内存。SLUB 通过减少 CPU 间的锁竞争和优化内存访问路径，显著提升了性能，特别适合频繁的小对象分配。

总的来说，使用虚拟内存进行访问的步骤如下所示：

- 分配页表所在内存空间并初始化页表
- 设置好基址寄存器（指向页表起始地址）
- 刷新TLB