练习1：理解first-fit 连续物理内存分配算法（思考题）

first-fit 连续物理内存分配算法作为物理内存分配一个很基础的方法，需要同学们理解它的实现过程。请大家仔细阅读实验手册的教程并结合kern/mm/default_pmm.c中的相关代码，认真分析default_init，default_init_memmap，default_alloc_pages， default_free_pages等相关函数，并描述程序在进行物理内存分配的过程以及各个函数的作用。 请在实验报告中简要说明你的设计实现过程。请回答如下问题：

你的first fit算法是否有进一步的改进空间?

物理内存分配的过程

1. 内存初始化（`default_init`）

- **过程**：在系统启动时，调用 `default_init` 进行物理内存管理的初始化。
- **作用**：此函数初始化内存管理器的相关数据结构，例如空闲页框链表，使系统了解当前内存状态，准备好分配和释放内存。

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

2. 内存映射初始化（`default_init_memmap`）

- **过程**：在内存初始化过程中，通过 `default_init_memmap` 函数初始化物理内存页框的信息。
- **作用**：此函数接受一个 `Page` 数据结构数组，表示一段物理内存，并将这些页框标记为可用或不可用。通过建立内存映射，系统能够区分已分配和空闲的内存块。
- **分配空闲页框**：初始化后，所有空闲页框形成一个链表，用于后续分配。

```
default_init_memmap(struct Page *base, size_t n) {  //基地址和物理页数量
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {                   //处理每一页
        assert(PageReserved(p));                    //是否为保留页
        p->flags = p->property = 0;                 //标志位清零，空闲块数量
        set_page_ref(p, 0);                         //将引用此物理页的虚拟页的个数清0
    }
    base->property = n;                             //开头页面的空闲块数量设置为n
    SetPageProperty(base);

    nr_free += n;                                   //新增空闲页个数nr_free
    list_add_before(&free_list, &(base->page_link));//将新增的Page加在双向链表指针中
}
```

3. 分配页框（`default_alloc_pages`）

- **过程**：当需要分配内存时，调用 `default_alloc_pages` 查找一个足够大小的空闲页框。
- **作用**：此函数遍历空闲页链表，寻找第一个满足请求大小的块，并将其从空闲链表中移除，返回分配的页框地址。采用 First Fit 策略，即找到的第一个合适的块立即分配，以保证分配效率。
- **更新内存状态**：分配后，系统会更新内存管理器中的页框状态，确保分配的信息准确记录。

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

4. 释放页框（`default_free_pages`）

- **过程**：当不再需要使用某块内存时，调用 `default_free_pages` 释放页框。
- **作用**：此函数接收一个 `Page` 结构，表示将某块页框归还给内存管理器。释放时，会将页框重新加入空闲链表，使其可以再次分配。
- **碎片处理**：在释放过程中，有些实现会尝试合并相邻的空闲块，以减少碎片化现象，但 First Fit 的标准实现通常不会合并，主要依赖后续的改进优化策略。

```
static void
default_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);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    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));
            }
        }
    }

    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}
```

总结

在 First Fit 算法的实现中，系统按以下顺序完成内存分配：

1. 初始化内存管理器和空闲链表。
2. 映射和标记空闲页框区域。
3. 分配时，从头查找空闲链表中的第一个合适块，找到后分配并更新链表。
4. 释放时，将页框归还空闲链表，允许重新分配。

First Fit 算法的简单结构确保了快速分配，但在频繁的分配和释放中会产生碎片问题，可通过后续的优化策略来缓解。

进一步的改进空间——改进为Next Fit算法

即每次从上一次分配结束的位置继续查找。这种策略可以避免频繁的从头开始扫描，减少遍历时间。

Next Fit 是一种内存分配算法，类似于 First Fit，但它在寻找可用内存块时略有不同。它在内存分配过程中，从上次分配的位置继续搜索下一个可用的内存块，而不是每次都从链表的头部开始，下面是对它的介绍。

工作原理

1. **初始化**:
   - 系统启动时，初始化内存管理器，标记所有物理内存页框的状态（可用或已分配），并将空闲块组织成链表。
2. **分配内存**:
   - 当请求分配内存时，Next Fit 算法从上次分配结束的位置开始搜索空闲链表。
   - 如果找到一个足够大的空闲块，它就会分配这块内存，并更新指针指向下一个空闲块。
   - 如果当前块不够大，则继续在链表中查找，直到找到合适的块，或者循环回到链表的开头，继续查找。
3. **释放内存**:
   - 当某块内存不再需要时，调用释放函数将其标记为可用，并将其加入到空闲链表中。
   - 释放后的内存块可能会与周围的空闲块合并（如果有实现合并逻辑的话），以减少内存碎片。

优缺点

优点

- **速度较快**：因为 Next Fit 从上次分配的位置开始查找，这样可以减少从头部开始搜索的时间，尤其是在大量内存请求的情况下，能提高效率。
- **减少内存碎片**：在某些情况下，Next Fit 可能会导致比 First Fit 更少的内存碎片，因为它的搜索方式更接近内存使用模式。

缺点

- **可能的碎片问题**：尽管 Next Fit 有时会减少碎片，但在长时间运行后，仍然可能产生大量的小碎片，尤其是在频繁的分配和释放过程中。
- **不一定全局最佳**：Next Fit 只考虑最近分配的块，可能会忽视链表头部的一些合适块，导致效率不如某些其他算法（如 Best Fit）。

除了改进为Next Fit算法外，还能进行一下改进

1. **合并空闲块**：在释放内存时，可以增加逻辑来合并相邻的空闲块，以减少碎片。
2. **动态调整搜索策略**：可以考虑动态调整起始搜索位置，结合其他算法的特点来优化内存分配过程，比如根据使用情况调整 Next Fit 的搜索起始点。
3. **使用空闲链表**：为了更好地管理内存碎片，可以引入分级链表或使用多链表来追踪不同大小的空闲块，以便更高效地找到合适的块。
4. **结合使用其他算法**：可以考虑与其他算法（如 Best Fit 或 Buddy 系统）结合使用，选择合适的场景下采用不同的分配策略。

扩展练习Challenge：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，请问你有何办法让 OS 获取可用物理内存范围?

办法：

1. BIOS/EFI 内存映射表

BIOS 或 UEFI 提供了系统内存映射表（Memory Map），包含硬件的物理内存布局信息。操作系统在引导过程中可以从 BIOS/EFI 中读取该表，确定可用的物理内存区域和保留区域（例如，内核和设备内存）。

2. 内存自检

操作系统可以在初始化阶段通过内存自检方法（Memory Probing）确定物理内存范围。具体做法是尝试向不同的物理地址写入和读取数据，并检查是否成功，以此确定可用的内存区域。但这种方法会有风险，因为有些地址可能已经被硬件保留。

3. ACPI 表获取内存信息

ACPI（Advanced Configuration and Power Interface）提供了内存信息表格，可以帮助操作系统了解系统硬件的可用物理内存范围。通过 ACPI 表，OS 能够识别内存分布和设备使用情况，确定哪些内存块是可用的。

4. 多级引导加载程序（如GRUB）

操作系统可以依赖引导加载程序（如 GRUB）来获取物理内存信息。GRUB 等引导加载程序在加载操作系统之前，已经从 BIOS/EFI 读取了内存信息，并将此信息传递给操作系统，从而帮助 OS 获知可用物理内存范围。

5. NUMA 配置

对于多处理器系统，NUMA（Non-Uniform Memory Access）配置可提供物理内存的节点级别信息，帮助 OS 确定不同内存块的可用性和分布。

结合这些方法，操作系统可以灵活地获取当前硬件的可用物理内存范围，确保系统内存资源的高效管理和合理分配。