练习一

回答将以物理内存的分配过程为主线，解析过程中遇到的各类函数的作用。

default_init

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

该函数是将链表freelist初始化为空，即前缀后缀都指向自己，并记录空闲页数为0。

default_init_memmap

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 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));
            }
        }
    }
}
```

该函数扫描了从base到base+n的地址内的空闲页，将其合为一个空闲块。具体实现：

1. 检查约束：不允许把长度为 0 的区间注册为空闲块。这是基本前置条件。
2. 指针指向base，然后循环遍历这n个page，标记为reserved，清楚标志位和property为0，base位置的property为n表示块大小，每个页的引用计数清零。
3. 全局空闲页加n，将当前的块插入空闲列表中，base比较每个节点的地址，找到第一个大于base的（之前都小于）节点，使用add_before函数插入。

default_alloc_pages

```
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);

    struct Page *p = list_next(&free_list);
    while (p != &free_list) {
        if (base + n <= p) break;
        p = list_next(p);
    }
    list_add_before(p, &(base->page_link));
    nr_free += n;

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

    p = list_next(&(base->page_link));
```

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

first-fit算法的核心函数，分配空闲块/页，函数原型中的n是期望分配页数，然后函数主体就是遍历空闲页表，寻找第一个符合条件的块（块大小>=n），如果刚好够就直接分配，如果大于，则把分配完剩余的空闲页数分出来作为新的空闲块放回应该在的位置。

default_free_pages

```
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);

    struct Page *p = list_next(&free_list);
    while (p != &free_list) {
        if (base + n <= p) break;
        p = list_next(p);
    }
    list_add_before(p, &(base->page_link));
    nr_free += n;

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

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

- 循环处理 base 开始的每一页。
- 保证每一页不是“保留页”（例如内核自身使用的页）；
- 保证每一页不是“属性页”（说明这段页之前没被标记为空闲块）；
- 然后清除标志位，重置引用计数为 0。
- 在空闲链表中找到合适的插入点（按照地址顺序）。
- 把当前空闲块插入进去。
- nr_free 是空闲页总数，更新它。
- 查看空闲链表中 base 前一个空闲块；
- 如果前一块的结束地址刚好接上 base（说明连续），就合并它们：
- 把前一块长度加上当前块；
- 删除当前块的节点；
- 清除当前块的“空闲头”标记。

操作系统回收一段刚用完的内存，把它重新拼接进空闲内存表中，尽量和前后的空闲块合并成更大的连续空闲区。

first-fit改进

1. **记录上次分配位置：**当前算法是每次要分配空间都从头开始遍历，改进就可以记录下上次遍历到的位置，下次从这里开始，避免频繁从头扫描，提高平均查找效率，代价是多了一个记录上次扫描位置的指针
2. **按块大小分为多组空闲链表：**

   ```
   list_head free_list_small;
   list_head free_list_medium;
   list_head free_list_large;
   ```

   分级first-fit也可以减少遍历次数。
3. **延迟合并**：不是每次都立即合并，而是将合并延后到下一次分配前，减少频繁操作链表的开销。
4. **维护空闲区间:**用区间树（如红黑树）记录空闲内存区间，插入/删除/合并更高效。

练习2

设计实现过程

1.空闲块管理结构

代码用 struct Page 作为物理页描述符，每个空闲块的首页 property 字段记录连续空闲页数。所有空闲块通过 free_list 链表组织。

2.初始化

best_fit_init 用 list_init(&free_list) 初始化链表，nr_free = 0。best_fit_init_memmap 遍历每个页框，清空标志和引用计数，只在首页设置 property = n，并插入链表（按地址有序）。

3.分配算法（best_fit_alloc_pages）

遍历 free_list，查找 property >= n 且最小的块（best-fit）。

找到后，若块大小大于 n，则将剩余部分重新插入链表（list_add），并更新 property。

分配后用 ClearPageProperty(page) 清除分配块的属性标记，nr_free -= n。

```
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;
    }
}
```

4.释放算法（best_fit_free_pages）

释放时，先清空页标志和引用计数，然后设置 property = n，插入链表（按地址有序）。

检查前后相邻块是否连续，若连续则合并（更新 property，删除被合并块）。

合并用 ClearPageProperty 和 list_del 实现。

```
le = list_prev(&(base->page_link));
while (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_prev(&(base->page_link));
    } else {
        break;
    }
}
```

5.测试与验证

best_fit_check 自动测试分配和释放逻辑，确保算法正确。

物理内存分配与释放原理

分配时，始终选择最适合（最小但足够）的空闲块，减少大块被小请求切割造成的碎片。

释放时，合并相邻空闲块，进一步减少碎片，提高大块分配成功率。

所有操作均通过链表维护，保证遍历和插入效率。

有进一步改进空间

1. 提高分配效率

每次分配都要遍历整个链表，时间复杂度 O(N)。

改进：

可用平衡二叉树（如红黑树）或跳表管理空闲块，按块大小排序，查找最合适块只需 O(logN)。

或维护多个链表，每个链表存放不同大小区间的空闲块（分级链表），小块/大块分配更快。

2. 减少内存碎片

释放时只合并相邻块，碎片仍可能较多。

改进：

可采用伙伴系统（Buddy System），释放时自动合并同级伙伴块，碎片更少，分配/释放效率高。

或定期整理空闲块（如后台合并），进一步减少碎片。

3. 支持多线程/并发

链表操作未加锁，多核环境下可能出错。

改进：

在分配和释放时加锁（如自旋锁），保证并发安全。

或采用 lock-free 数据结构，提升多核性能。

4. 分配策略优化

只实现了 best-fit，部分场景下可能导致小块大量残留。

改进：

可结合 next-fit、worst-fit 等策略，根据实际负载动态切换分配算法。

或引入延迟合并、预分配等机制，提升大块分配成功率。

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，可以通过下面方法让 OS 获取可用物理内存范围：

1. 读取硬件启动信息

x86平台（BIOS/UEFI e820表）

BIOS/UEFI在启动时会生成e820内存映射表，描述所有物理内存段的起始地址、长度和类型（如可用、保留、设备等）。

1. 操作系统启动时通过中断或固件接口读取e820表，遍历所有类型为“可用”的区域，作为物理内存管理的基础。
2. 例如Linux内核启动时会解析e820表，构建物理内存页框管理结构。

ARM/RISC-V平台（设备树DTB）

1. 嵌入式和RISC-V等平台常用设备树（Device Tree Blob, DTB）描述硬件资源。DTB中有“memory”节点，包含物理内存的起始地址和大小。
2. 内核启动时解析DTB，获得可用物理内存范围。

```
mem_begin = get_memory_base_from_dtb();
mem_size  = get_memory_size_from_dtb();
```

2. Bootloader传递参数

Bootloader（如GRUB、U-Boot）在加载内核前会检测物理内存，并通过命令行参数、环境变量或专用结构体（如multiboot info）将内存信息传递给内核。

内核启动时读取这些参数，初始化物理内存管理。例如，GRUB会将内存映射表传递给Linux内核，内核据此初始化页框。

3. 硬件寄存器或MMIO接口

某些平台（如部分嵌入式SoC）有专用寄存器或MMIO区域，存储物理内存大小和分布。内核可通过访问这些寄存器获取信息。

例如部分ARM芯片的系统控制器会提供内存大小寄存器，内核读取后即可确定可用物理内存范围。

4. 虚拟化环境接口

在虚拟机中，Hypervisor（如KVM、QEMU）通过ACPI、virtio等标准接口向客户操作系统提供物理内存信息。

客户操作系统通过这些接口获取可用物理内存范围，适配不同虚拟机配置。