Lab2

杨竣羽　何潇　王豪杰

练习1 理解first-fit 连续物理内存分配算法（思考题）

实验目的：

- 理解页表的建立和使用方法 （CTRL shift +】）
- 理解物理内存的管理方法
- 理解页面分配算法
- 实现不同的物理内存分配算法

实验要求：

- 了解如何发现系统中的物理内存，然后学习如何建立对物理内存的初步管理，即了解连续物理内存管理，最后掌握页表相关的操作。
- 主要是在实验一的基础上完成物理内存管理，并建立一个最简单的页表映射。

实验原理

- **内存管理原理**

本实验中无论什么内存分配算法，其底层逻辑都是一样的，所以我认为操作系统内存管理的原理是本次实验重要的知识点之一。在ucore指导书中详细解释了内存管理的基本原理：

基本概念：

- 页（Page）：操作系统中内存分配的最小单位。
- 页的管理结构（Struct Page）：每个页都有其对应的结构体，用来描述页的状态：

```
struct Page {
    unsigned int flags;         // 标志位（是否空闲、是否保留等）
    unsigned int property;      // 若是空闲块头页：记录连续页数
    list_entry_t page_link;     // 把当前页（块头）挂在 free_list 上
    int ref;                    // 引用计数
};
```

- 空闲块（free block）：指若干连续的空闲页。
- 空闲块链表（free_list链表）：一个循环双向链表。在初始化阶段后，符合条件的空闲块会被插到上面，同时链表上记录了所有空闲块的位置和大小。

基本阶段：

- 初始化阶段：把所有物理内存映射为页，找出“已保留页”，创建全局空闲页链表。
- 分配阶段：根据用户的需求（n字节或n页等）查找空闲链表中的合适块（各算法最大的区别就是查找合适块的方法不同），分配该块，同时更新数据。
- 释放阶段：把释放块重新加入空闲链表

代码分析：

first-fit主体部分：

几个关键宏

```
assert(p)：用于判断执行结果是否为真。
PageReserved(p)：判断页面是否被系统保留，不能释放。
PageProperty(p)：判断页面是否有 property 属性（即是否在空闲块头）。
set_page_ref(p, val)：设置页面引用计数。
SetPageProperty(p) / ClearPageProperty(p)：设置或清除 property 标志。
list_add, list_add_before, list_del, list_next：双向链表操作。
le2page(le, member)：通过链表节点获取对应的 Page 指针。
```

default_init函数：

```
static void
default_init(void) {
    list_init(&free_list);  //初始化空表表头。free_list 是一个全局的双向循环链表头结点，用来存放所有的“空闲物理页块”
    nr_free = 0;            //无可用的空闲页
}
================================================================================================
================
#list_init函数的定义
static inline void
list_init(list_entry_t *elm) {   //list_entry_t是一个典型的双向链表节点结构
    elm->prev = elm->next = elm;   //prev和next分别指向前一个和后一个节点。这句话意思是让链表的前驱和后驱都指向自己。
}
```

list_init函数中elm既是链表的起点、又是链表的终点。这样做构成了一个“空链表”的初始化状态。于是在插入、删除节点时无需判断“空链表”，避免空指针错误。

default_init函数中调用list_init函数，初始化空闲页链表，**把空闲页链表的头准备好**，为后续内存分配算法打基础。

default_init_memmap函数：（内存管理初始化的关键阶段）

```
static void
default_init_memmap(struct Page *base, size_t n) {    //base：一片连续物理页的起始地址（第一个 Page 结构体   n：这片区域中的页数。
    assert(n > 0);     //用assert函数判断要初始化的页数是否大于0，确定需要初始化的页数不为0。
    struct Page *p = base;  //创建一个指向page结构体的指针p，将指针p初始化为指向base的初始内存地址。
    for (; p != base + n; p ++) {     //for循环遍历每个页面。
        assert(PageReserved(p));              //用PageReserved()宏找出“已保留页”，因为在系统启动时，所有未使用的物理页都会被标记为“保留”，只有标记为“保留”的页，才能被重新初始化成“空闲页”。
        p->flags = p->property = 0;  //清除原有标志位flags，清空页的属性信息property。
        set_page_ref(p, 0);    //此时没人占用这个页。
    }
    base->property = n;  //在第一个页记录这个块的大小（包含 n 个连续页）a
    SetPageProperty(base);    //设置该页的「Property 位」，标明这是一个空闲块的头页（其余页是从属）
    nr_free += n;    //此时空闲页增加了n页
    if (list_empty(&free_list)) {      //若free_list是空的。
        list_add(&free_list, &(base->page_link));      //把base这个新块作为第一个节点插入。
    } else {    //若free_list非空，则从链表头开始遍历每个节点，遇到链表末尾的free_list（因为是循环链表）为止
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {    //若要插入的base地址比当前节点page小，说明应该插在这个节点前面。
                list_add_before(le, &(base->page_link));    //把base放在page前。
                break;
            } else if (list_next(le) == &free_list) {    //此时le是链表的最后一个节点，若遍历到最后都没找到大于base的块，就把base插到链表末尾。
                list_add(le, &(base->page_link));    //即把base插到节点le之后，free_list之前。
            }
        }
    }
}
============================================================
```

在default_init_memmap（base,n）执行时，会初始化一段连续的新空闲的内存[base,base+n]，同时把他们设置成空闲页，并将这个空闲块插入全局空闲页链表（free_list）中，具体操作流程在函数注释中详细说明。

default_init_memmap函数在内核中的作用：该函数通常出现在操作系统的物理分配器初始化阶段。系统启动时，在检测到哪些页可用之后会调用该函数，把这些区间标记为空闲。为随后的内存分配/回收做准备。

default_alloc_pages函数：（实际的物理页分配逻辑）

```
static struct Page *
default_alloc_pages(size_t n) {    //从该部分开始正式涉及first-fit算法，即选择第一个能满足请求的空闲块。n表示要分配的连续页数。
    assert(n > 0);      //尝试从空闲页链表中分配的连续n'页物理页数要大于0，即非空。
    if (n > nr_free) {
        return NULL;
    }           //要保证请求的页数小于系统的空闲页数。
    struct Page *page = NULL;     //定义一个page，用于保存最后被选中的空闲块的头页指针。起始设置为null；
```

```
    list_entry_t *le = &free_list;         //定义链表节点指针le，指向链表头。
    while ((le = list_next(le)) != &free_list) {  //遍历整个空闲块链表
        struct Page *p = le2page(le, page_link);  //根据最后面描述的le2page的宏定义，知道此时p 就是当前空闲块的起始页。
        if (p->property >= n) {    //这个就是first-fit策略。若当前链表的空闲页大于要分配的连续页数n，即找到了够大的块。
            page = p;    //用page作为找到的空闲块。
            break;
        }
    }
    if (page != NULL) {     //此时上文定义的p全权换为page。找到够大的空闲块之后。此时上文定义的p全权换为page。剩下要做的就是把找到的这块空闲块从空闲链表中分离出去。
        list_entry_t* prev = list_prev(&(page->page_link));    //找出page块的前驱节点prev。
        list_del(&(page->page_link));                  //用list_del把page块从空闲链表块摘除。
        if (page->property > n) {                //看到此处的时候我产生了一个疑问：上述还用p做起始页的时候已经有过判断 if (p->property >= n)，找到空闲块并选中为page，那为什么后面还有这行的page判断。我刚开始时觉得多此一举，但后面思考发现二者虽然判断条件差不多，但效果并不相同。第一个判断用来“查找”，即确定找到可用的块，然后停止遍历；而第二个判断用来“分配”，确定该块是否过大，需要拆分。具体操作看下文。
            struct Page *p = page + n;     //此时用p计算新的空闲块的起始页。
            p->property = page->property - n;    //计算把n页连续的页分配进page后还剩多少页
            SetPageProperty(p);                //将p设为空闲块的起点。
            list_add(prev, &(p->page_link));       //将剩余的块插回到链表，并且在“prev”之后，其中“prev”就是之前空闲块还未从链表删除时的前驱节点。
        }
        nr_free -= n;  //系统的空闲页数减少了使用的n页。
        ClearPageProperty(page);    //清除page中的空闲标志，表示页已经被占用，不属于空闲链表。
    }
    return page;
}
==========================================================================
    #define le2page(le, member) to_struct((le), struct Page, member)  //le2page的宏定义：表示取到拥有这个page的结构体。
```

default_alloc_pages函数实现了最简单的“首次适配”物理页分配算法：从空闲块列表中找到第一块足够大的连续空闲页，必要时拆分剩余部分，更新空闲统计，并返回分配得到的页指针。

default_free_pages函数：

```
static void
default_free_pages(struct Page *base, size_t n) {   //释放（回收）物理页块到空闲内存链表，即释放从base开始的n个物理页
    assert(n > 0);  //要释放的物理页数要大于0
    struct Page *p = base;    //用指针p代表起始地址
    for (; p != base + n; p ++) {      //从p=base开始，遍历[base,base+n)区间中的对象。
        assert(!PageReserved(p) && !PageProperty(p));    //确保释放的每页都是非保留的、非空闲块头的有效可用页。
        p->flags = 0;     //清除标志位flags，把该页变为空白页结构。
        set_page_ref(p, 0);    //用该宏设置该页面引用计数为0。即该页无人使用。
    }
    base->property = n;    //让base作为空闲块的“块头页”，大小为n。
    SetPageProperty(base);  //标记base为“property”
    nr_free += n;     //空闲页数增加了n页。
```

```
    //将新释放的部分插入空闲链表free_list

    if (list_empty(&free_list)) {   //若此时空闲链表为空
        list_add(&free_list, &(base->page_link));   //用list_add把新释放的页块插入全局空闲链表。
    } else {
        //若空闲链表为非空。
        list_entry_t* le = &free_list;    //用le做空闲链表的游标。
        while ((le = list_next(le)) != &free_list) {   //遍历整个空闲链表。
            struct Page* page = le2page(le, page_link);   //用le2page宏将当前链表节点le转换为对应的page对象，便于后续访问页面的物理序信息。
            if (base < page) {   //若要插入的块base在当前节点page之前
                list_add_before(le, &(base->page_link));    //把当前块插入到page前
                break;
            } else if (list_next(le) == &free_list) {  //当遍历到最后也没有找到大于base的地址，则当前块在链表的末尾，所以直接将base插到表尾。
                list_add(le, &(base->page_link));
            }
        }
    }

    //default_free_pages()的核心部分，即为了避免外部碎片化，将base与其物理相邻的空闲块合并成更大的连续块。
    //尝试和前面的合并。
    list_entry_t* le = list_prev(&(base->page_link));   //用le做base的前一个节点。
    if (le != &free_list) {    //当base'的前一个节点不是头节点时，继续向下执行合并。
        p = le2page(le, page_link);   //将链表节点转回page实例。
        if (p + p->property == base) {  //若前块地址+前块页数==当前地址，即前一块和base'相邻
            p->property += base->property;   //先将二者的尺寸合并
            ClearPageProperty(base);   //再将base的property标志去掉，代表此时base'不再是块头
            list_del(&(base->page_link));  //从链表中删除base节点。代表其已经与前一个节点合并。
            base = p;  //此时更新base为新的合并块头。
        }
    }
   //尝试和后面的合并。
    le = list_next(&(base->page_link));    //用le做base的后一个节点。
    if (le != &free_list) {   //确保le之后至少有一个空闲块。
        p = le2page(le, page_link);    //用p表示还原后的page结构体。
        if (base + base->property == p) {   //若当前块刚好与后一块p相邻
            base->property += p->property;  //把二者的尺寸合并。
            ClearPageProperty(p);   //去掉p的property标准。
            list_del(&(p->page_link));   //删除后块p。
        }
    }
}

static size_t
default_nr_free_pages(void) {
    return nr_free;
//用来获取当前的空闲页面数量。
```

default_free_page(base,n)函数用于释放从base开始的n个连续物理页，把它们重新加入到空闲链表（free_list）中，并与符合条件的相邻空闲块合并，防止内存碎片化。

验证部分：

basic_check（）--基础正确性测试

```
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);
```

```
}
```

basic_check（）是用来验证基本内存分配/释放机制是否正确稳定的函数，主要是对alloc_page、free_page的验证，例如是否能分配出不同的页、分配出的页是否重复、状态是否正确、页物理地址是否越界等；

default_check（）--复杂的 first-fit 策略测试

```
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（）是“高级检查”部分，用于验证分配算法策略是否正确执行了 首次适应法 (first-fit) 原则。该部分是在basic_check（）的基础上，进一步验证first-fit策略是否正确、空闲块释放后能否合并以及重新分配的逻辑正确性。

default_pmm_manager（）--管理器结构

```
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,                       // 初始化物理内存系统
    .init_memmap = default_init_memmap,         // 初始化空闲页块（free list）
    .alloc_pages = default_alloc_pages,         // 分配页接口
    .free_pages = default_free_pages,           // 释放页接口
    .nr_free_pages = default_nr_free_pages,     // 查询空闲页接口
    .check = default_check,                     // 检测钩子（上述测试函数）
};
```

这是整个物理内存管理模块的“函数指针表”，定义一组统一接口供系统使用。

first-fit算法的改进空间：

- ## 外部碎片化问题

  问题描述：长期使用后，小块分配频繁，内存会变得零碎，导致大块申请失败。

  具体分析：内存碎片分为外部碎片和内部碎片。二者分别产生于不同的情况，内部碎片是由于分配的块比实际需求更大，导致块内浪费；外部碎片是由于零散的小空闲块分布在不同的位置，导致无法满足大块的需求。这样看来，虽然在first-fit中存在相邻内存块合并的算法，但不是所有的小内存块都是相邻的，小内存块不能跨越被占用的区域合并，所以会产生很多碎片化内存块。

改进方向：采用buddy system（按2的幂次划分块）；增加块合并策略。

- ## 分配不均衡问题

问题描述：长期从前部开始分配造成低地址区域频繁分裂，而后部保持大块。

具体分析：系统每次从头部开始寻找一块足够大的空闲块分配，导致低地址或前半部分的内存区域被频繁使用、切割；而高地址部分长期保持大块未使用。这样会导致低地址内存区中都是碎片化的空闲块，而高地址内存区中大块连续空闲块未被使用，资源分配不合理。

改进方向：

- 改用next-fit（下次适应）分配法。即保留一个“游标”记录上次分配结束位置，下次从这个位置继续向后找，到达链表尾后再从头开始循环。
- 引入 按大小分级的链表（Segregated List）。即小块/中块/大块分别管理，从不同链表分配，避免大块反复被切。