lab2实验报告

1.实验目的

理解页表的建立和使用方法

理解物理内存的管理方法

理解页面分配算法

2.实验内容

实验二主要涉及操作系统的物理内存管理。

操作系统为了使用内存，还需高效地管理内存资源。本次实验我们会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，即如何建立页表来实现虚拟内存到物理内存之间的映射，帮助我们对段页式内存管理机制有一个比较全面的了解。本次的实验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。

2.1 练习 0：填写已有实验

已经在lab1的实验报告和代码当中提交了相关展示，不做重复赘述。

2.2 练习 1：理解 first-fit 连续物理内存分配算法（思考题）

first fit的原理：

- 空闲分区列表按地支顺序排序
- 分配过程时，搜索一个合适的分区
- 释放分区时，检查是否可与临近的空闲分区合并

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，我们结合了 kern/mm/default_pmm.c 中的相关代码，认真分析 了default_init，default_init_memmap，default_alloc_pages，default_free_pages 等相关函数

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

这两个函数用于初始化空闲页链表,初始化每一个空闲页，然后计算空闲页的总数。 函数说明： 函数的两个参数：*base表示该空闲块的开始地址，n表示页数量 如果此页是空闲页，并且不是空闲块的第一页，则p->property应设置为0。如果此页是空闲页，并且是空闲块的第一页，则p->property应设置为total num of block。p->ref应该是0，因为现在p是空闲的，没有引用。

```
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));
            }
        }
    }
}
```

这是一个名为default_init_memmap的静态函数。逐步分析代码的作用：

1. 首先，函数断言参数n大于0，即保证n的值大于零。这是为了确保有要处理的页面存在。
2. 函数定义了一个指向Page结构的指针p，并将其初始化为参数base的值。接下来，通过循环遍历从base到base + n的所有页面。
3. 在循环内部，函数使用断言(PageReserved(p))来确保页面p是被保留的。
4. 然后，函数将页面p的flags和property字段设置为零，并调用set_page_ref函数将页面p的引用计数设置为零。
5. 在循环结束之前，将参数base页面的property字段设置为n，并调用SetPageProperty函数将页面base标记为具有特殊属性。接着，将n加到nr_free变量上，增加可用页面的数量。
6. 接下来，函数检查free_list是否为空。如果为空，将base页面添加到free_list中，使用list_add函数将base->page_link添加到free_list的头部。
7. 如果free_list不为空，则进入else语句块。函数定义了一个指向list_entry_t类型的指针le，并将其初始化为free_list的地址。然后，进入一个while循环。该循环通过调用list_next函数迭代遍历free_list中的每个元素，直到再次到达free_list。

8. 在每次循环迭代时，函数将当前元素le转换为Page结构的指针page（通过调用宏le2page(le, page_link)）。然后，它进行以下判断：
    - 如果base小于page，则通过list_add_before函数将base->page_link插入到le之前的位置，并使用break语句终止循环。这是为了确保free_list中的页面是按照地址从小到大的顺序排列的。
    - 否则，如果list_next(le) == &free_list，则通过list_add函数将base->page_link插入到le之后的位置。这是为了确保base是free_list中的最大地址页面，并且放置在free_list的末尾。
9. 最后，函数结束while循环后，返回到调用者。

综上所述，这段代码的作用是初始化一块内存页面映射。它将给定范围内的页面的属性、引用计数等字段设置为初始值，并将页面按照地址从小到大的顺序插入到free_list中。该函数假设页面已经被保留，并且free_list中的页面已按地址排序。

```
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;
}
```

下面是对提供的代码进行逐步解析的分析：

1. 首先，函数断言参数n大于0，以确保n的值大于零。
2. 接下来，函数检查请求分配的页面数量n是否大于可用的空闲页面数量nr_free。如果是，则返回NULL，表示没有足够的空闲页面可供分配。

3. 函数声明一个指向Page结构的指针page，并将其初始化为NULL。接着，定义一个指向free_list的list_entry_t类型指针le，并将其初始化为free_list的地址。

4. 进入while循环，该循环迭代访问free_list中的每个元素，直到再次到达free_list。在每次循环迭代中，函数将当前元素le转换为Page结构的指针p（通过调用宏le2page(le, page_link)）。

5. 在循环内部，函数检查页面p的属性（property字段）是否大于等于请求分配的页面数量n。如果是，则将page指针设置为p，并使用break语句终止循环。这意味着找到了一个具有足够空闲页面的Page结构。

6. 如果page指针不为NULL（即找到符合要求的Page结构），则继续执行下面的代码；否则，跳过下面的代码并返回NULL。

7. 在这一部分，函数首先获取page的前一个页面，并通过list_del函数从free_list中移除page（即将page->page_link从链表中删除）。接着，它检查page的属性是否大于请求分配的页面数量n。如果是，则需要拆分page，以便保留剩余的页面作为新的空闲页面。这样做可以减小原始页面的属性值，并将其添加到free_list中合适的位置。对于剩余的页面，会将其属性设置为剩余页面的数量，并使用SetPageProperty函数将其标记为具有特殊属性。最后，将剩余页面（指向的新Page结构）插入到prev指针所指向的元素之前。

8. 操作完页面后，将已经分配的页面数量n从nr_free变量中减去。之后，通过调用ClearPageProperty函数清除页面page的特殊属性标记。

9. 最后，返回分配的页面指针page，或者NULL（如果没有足够的空闲页面可供分配）。

综上所述，这段代码的作用是从空闲页面列表free_list中分配请求的页面数量n。它迭代查找具有足够空闲页面的Page结构，并在找到合适的页面后对其进行处理。如果成功分配页面，则相应地更新页面的属性，并返回分配的页面指针；否则，返回NULL表示。

```
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));
        }
    }
}
```

下面是对提供的代码进行逐步解析的分析：

1. 首先，函数断言参数n大于0，以确保n的值大于零。

2. 接下来，函数声明一个指向Page结构的指针p，并将其初始化为传入的参数base。之后，使用循环遍历从base到base + n的所有页面，对每个页面执行以下操作：

    - 使用断言函数assert检查页面p不是被保留的（Reserved）并且没有特殊属性（Property）。如果断言条件不满足，将会触发断言错误。
    - 将页面的flags字段设置为0，即清除页面的标志位。
    - 使用set_page_ref函数将页面的引用计数设置为0，表示该页面没有被引用。

3. 在循环之后，设置base页面的属性(property字段)为请求释放的页面数量n，并使用SetPageProperty函数将其标记为具有特殊属性。

4. 将nr_free变量增加n，表示增加了n个空闲页面的数量。

5. 接下来，检查空闲页面列表free_list是否为空。如果是空的，将base页面插入到free_list的头部。

6. 如果free_list不为空，则遍历free_list中的每个元素，直到再次到达free_list。在循环中，将当前元素le转换为Page结构的指针page（通过调用宏le2page(le, page_link)）。

    - 检查base页面是否小于当前页面page。如果是，将base页面插入到当前元素le之前，并使用break语句终止循环。
    - 否则，如果没有找到比base页面大的页面，将base页面插入到free_list的尾部。

7. 找到base页面的前一个页面，将其赋值给指针le。然后，检查le是否不等于free_list，即确保找到了前一个页面。
    - 如果找到了前一个页面，则将前一个页面p与其相邻的property相加，以合并相邻的空闲页面。该合并操作将base页面的property字段增加，并将base页面从链表中删除，然后将base指向前一个页面p。
    - 如果没有找到前一个页面，则跳过合并操作。
8. 找到base页面的后一个页面，将其赋值给指针le。然后，检查le是否不等于free_list，即确保找到了后一个页面。
    - 如果找到了后一个页面，则将base页面的property字段增加相应的页面数量，并将后一个页面p从链表中删除。该操作用于合并相邻的空闲页面。
    - 如果没有找到后一个页面，则跳过合并操作。

综上所述，这段代码的作用是释放指定数量的页面（从base开始的n个页面）。它首先对每个页面清除标志位和引用计数，然后更新页面的属性，并标记为具有特殊属性。然后，它根据页面的位置在空闲页面链表中找到适当的位置，并将页面插入到链表中。最后，它尝试合并相邻的空闲页面，以减少空闲页面的碎片化。

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。

这段代码是一个物理内存分配的过程，主要用于管理和操作空闲页面列表。

程序的设计思路如下：

1. 首先，将一段连续的物理页面作为空闲页面传入函数，由参数base表示。这些页面可能是操作系统在启动时分配的未被使用的物理页面。
2. 函数会遍历这些页面，对每个页面执行以下操作：
    - 检查页面的保留状态和特殊属性是否满足要求，若不满足将会触发断言错误。
    - 清除页面的标志位和引用计数，以确保页面不被使用和标记为可用。
3. 在循环完成后，函数将设置第一个页面(base)的property字段为请求释放的页面数量n，并标记该页面具有特殊属性。
4. 接下来，函数会根据空闲页面链表free_list的情况，将当前页插入到相应的位置。如果链表为空，直接将当前页面插入到链表头部；如果不为空，则需要在链表中找到合适的位置插入。
5. 在插入页面后，函数会尝试合并相邻的空闲页面。它首先找到当前页面的前一个页面，如果存在并且与当前页面相邻，将合并两个页面的property字段，并将当前页面从链表中删除。接着，函数找到当前页面的后一个页面，如果存在并且与当前页面相邻，将合并两个页面的property字段，并将后一个页面从链表中删除。

为了实现上述设计思路，代码中使用了一些辅助函数和宏：

- assert：用于进行断言检查，确保页面的保留状态和特殊属性满足要求。
- PageReserved()和PageProperty()：用于检查页面是否被保留或具有特殊属性。
- set_page_ref()：用于设置页面的引用计数。

- SetPageProperty()和ClearPageProperty()：用于设置或清除页面的特殊属性。
- list_empty()：用于检查链表是否为空。
- list_add()：将元素插入到链表头部或两个元素之间。
- list_entry_t、list_next()和list_prev()：用于遍历链表。
- le2page()：将链表元素转换为页面结构的指针。

综上所述，该程序通过管理空闲页面列表来实现物理内存分配和释放的功能，同时尝试合并相邻的空闲页面以减少内存碎片化。

对first fit算法的进一步改进

对于First Fit算法的进一步改进，以下是几种方法：

1. 优化分区选择：在找到合适的分区时，选择一个最接近所需大小的分区，而不是第一个可用的分区。这样可以减少碎片化。
2. 分区合并：在释放分区时，检查相邻的空闲分区是否可以合并成一个更大的分区。这样可以减少内存碎片。
3. 分区拆分：如果一个分区比所需大小大很多，可以将该分区拆分为两个部分，一个用于满足当前请求，另一个保留为未分配的分区。
4. 空闲分区排序：维护一个空闲分区列表，并按照分区大小进行排序。这样可以更快地找到合适的空闲分区，并减少搜索时间。
5. 动态分区大小调整：根据实际需求动态调整分区大小。如果一个分区长时间没有被使用或频繁分配释放，可以考虑缩小该分区的大小以提高内存利用率。

这些方法可以帮助改进First Fit算法，在一定程度上减少内存碎片，并提高内存利用率。但需要根据具体情况选择合适的优化策略。

2.3 练习 2：实现 Best-Fit 连续物理内存分配算法（需要编程）

在完成练习一后，参考 kern/mm/default_pmm.c 对 First Fit 算法的实现，编程实现 Best Fit 页面分配算法，算法的时空复杂度不做要求，能通过测试即可。

设计实现过程

best fit算法和first fit算法在本次的代码实现上，唯一的不同就是alloc函数的设计存在一定的区别。 在first fit算法中，我们选中的是从空闲页块链表中找到的第一个适合大小（大于所需空间）的空闲页块；而在best fit算法中，选中的是从空闲页块链表中找到的大于所需空间的最小的空闲页块，因此需要改动的部分是：

```
static struct Page *
best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
```

```
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
     /*------改动开始------ */
    size_t min_size = nr_free + 1;
    // 设置一个min_size变量来记录当前找到的 大于所需空间且最小的空闲页块的大小，其中
nr_free为当前空闲块的个数
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_size) {
            // 如果page的property属性（大小）满足
            // 1.大于所需要的块数
            // 2.小于当前找到的满足条件的最小块数
            page = p;
            min_size = p->property;
            // 更新min_size
        }
    }
     /*------改动结束------ */
    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;
}
```

另外，我们还需要在pmm.c中加上：

```
#include <best_fit_pmm.h>
```

并且在init_pmm_manger处修改管理算法为best fit:

```
static void init_pmm_manager(void) {
    pmm_manager = &best_fit_pmm_manager;
    // 修改↑
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

Best-Fit 算法的进一步的改进空间？

1. 时间复杂度角度

- 算法的时间复杂度是我们最先关注到的事情，因为在寻找最合适的空闲页块时，当前实现需要遍历整个空闲页链表。显然，遍历会导致一定的性能问题，在空闲页较多时，效率为O(n)。因此可以考虑使用更高效的数据结构，例如二叉搜索树或红黑树，来快速找到最佳匹配。 二叉搜索树的搜索操作最好的时间复杂度为O(logN)，最坏为O(N)，整体时间复杂度介于二者之间。比如定义这样的结构体，代表一个二叉搜索树的节点：

```
struct TreeNode {
    size_t size;
    struct Page *page;
    struct TreeNode *left;
    struct TreeNode *right;
};
```

- 红黑树是一种自平衡的二叉搜索树，它在插入和删除操作后会自动保持平衡，因此在平均情况下具有更好的性能，搜索的复杂度介于O(logN)和O(2logN)之间。在使用红黑树时，节点结构可能与上述二叉搜索树相似，但红黑树节点还需要包括红黑标志以维护平衡。

```
struct RBTreeNode {
    size_t size;
    struct Page *page;
    int color; // 0 : black, 1 : red
    struct RBTreeNode *left;
    struct RBTreeNode *right;
    struct RBTreeNode *parent;
};
```

2. 算法本身优化：

- 当需要分配多个页面时，当前实现只会选择一个最佳匹配的页面，并不会考虑合并相邻的空闲页面以减少碎片。
- 举例来说，按照顺序，我们分别有property为12、13、50的页块，我们需要的大小是25。best fit算法为我们选择的是50大小的页块，但实际上可以将12和13这两个合并，以最大程度地减少碎片。
- 或者说，在特殊情况下，当没有足够的空闲页面可供分配时，代码返回NULL。虽然这是合理的，但是可以考虑添加一些错误处理机制，比如上面提到的碎片合并。