

# lab2

## 操作系统课程实验报告

### 实验名称：Lab2 物理内存和页表

小组成员：2112849唐静蕾 2113388 高涵 2113999 陈翊炀

#### 一、实验目的

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。

#### 二、实验内容

##### 练习1:理解first-fit 连续物理内存分配算法

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？

首先笼统讲一下first-fit算法，作为连续物理内存分配算法的一种，就是在物理内存分配时，将空闲内存块按照地址从小到大的方式连起来，具体实现时使用了双向链表的方式。当分配内存时，从链表头开始向后找，这意味着从低地址向高地址查找，一旦找到可以满足要求的内存块，即将该内存块分配出去即可。

接下来看它的实现过程中具体涉及到的函数：

##### **default\_init:**

包含两条指令：

```
1 list_init(&free_list);
2 nr_free = 0;
```

首先初始化free\_list链表，这个链表是将内存中所有的空闲内存块通过链表的形式组织起来，就形成了最基础的free\_list，其具体初始化list\_init函数包含一条指令 `elm->prev = elm->next = elm`；作用是初始化链表，因为free\_list是个双向链表，所以要初始化前向后向指针；

nr\_free可以理解为在这里可以使用的一个全局变量，记录可用的物理页面数，此处初始化为0。

## default\_init\_memmap:

这段代码的两个参数分别是内存映射区域的起始位置base和连续物理内存区域的大小n；作用是初始化一块内存映射区域。具体步骤如下：

首先，有一个断言，代码会判断参数n（表示内存页数量）是否大于0，如果不大于0，则会触发断言错误；然后，定义一个指针p，指向内存映射区域的起始页（base）；接着，使用一个循环遍历内存映射区域的每一页，直到遍历到最后一页（base + n）；在每一页遍历过程中，首先会断言该页被保留（即PageReserved(p)返回true，该页被保留待用），如果不满足该条件，则又会触发断言错误；然后，将该页的flags和property字段初始化设置为0，其中flags记录当前page frame状态，property字段表示该段内存空间为连续且未被占用的空间；调用set\_page\_ref函数将该页的引用计数设置为0，表示当前没有引用指向该页。

将base页的property字段设置为n，表示整个内存映射区域的页数。

使用SetPageProperty将base页的属性设置为PageProperty。

将nr\_free增加n，表示空闲页的数量增加了n，也就是对应的内存可分配区域的大小。

接下来是将base页插入到空闲页链表（free\_list）中的适当位置。注意这个free\_list的维护是完全按照内存地址大小排序，与空闲块大小没有直接关系。首先判断空闲页链表是否为空，如果为空，则直接将base页插入到链表头部；如果空闲页链表不为空，则需要遍历链表找到合适的位置插入base页。使用一个指针le指向链表头部，然后进入一个循环，不断向后遍历链表（在遍历链表的过程中，获取当前节点对应的Page结构体指针page）。如果base页的地址小于page页的地址，则说明找到了合适的位置，将base页插入到当前节点的前面，并退出循环；如果当前节点的下一个节点是链表头部节点，说明已经遍历到了链表的末尾，将base页插入到链表尾部，并退出循环。

最后，返回到调用该函数的地方，完成初始化内存映射区域的操作。

## default\_alloc\_pages:

这段代码的一个参数是进程所需的连续物理内存区域的大小n；作用是分配一块大小为n内存区域给进程。具体步骤如下：

首先还是一个断言，这个断言表明进程所需的内存空间是大于0的，之后判断n与nr\_free的大小关系，上文可知nr\_free存储的是内存中所有可分配区域的大小，如果n>nr\_free表示进程所需空间过大无法得到分配，函数结束。

否则，初始化page指针为NULL，le指针指向free\_list空闲页链表，由于first-fit是按照内存地址大小排序查找，因此进行while循环查找，判断条件为(le = list\_next(le)) != &free\_list，即未完成free\_list遍历回到链表头时，利用函数le2page将指针切换到page头，判断如果当前空闲空间p->property >= 进程所需空间 n，内存分配成功，break跳出循环。

下面还要进行链表的维护，如果page不为空，即表示成功分配到内存，之后获取page链表中page的前一个页面块的指针，并将其赋值给prev。从page链表中删除page页面块。如果page的属性值大于n（表示当前页可用内存大于进程所需内存，即用不完的状态），则执行以下操作：声明一个指针p，指向page页面块的后面第n个页面块。将p的property属性值设为page的property属性值减去n，表示当前块可用内存减少了n（已被分配出去），之后设置属性，维护链表。

最后在内存中所有可分配区域的大小nr\_free中减去已经被分配的n，返回被分配的page指针。

## default\_free\_pages:

这段代码的两个参数分别是内存映射区域的起始位置base和需要释放的连续物理内存区域的大小n；作用是释放一块大小为n的进程使用完毕的内存区域。具体步骤如下：

首先是个断言，来确保传递给函数的n参数大于0，即要释放的页面数量必须是正数，否则会导致程序终止。创建一个指向struct Page类型的指针变量p，并将其初始化为传递给函数的base参数。接下来是一个for循环，它用于遍历从base开始的n个页面，在每次循环迭代中，使用断言来确保当前页面p既不是保留页面（PageReserved）也不是特殊属性页面（PageProperty）。如果这两个条件中的任何一个为真，将导致程序终止。将当前页面p的标志（flags）设置为0，并调用set\_page\_ref函数将页面的引用计数（page\_ref）设置为0，循环结束。将基页面（base）的property字段设置为n，然后将基页面标记为特殊属性（SetPageProperty），将全局变量nr\_free增加n，表示空闲页面的总数量增加了，

检查空闲页面链表free\_list是否为空。如果为空，将基页面base添加到空闲页面链表的末尾，使其成为链表中的第一个元素，如果空闲页面链表不为空，创建一个指向链表头的指针le。在一个循环中，迭代遍历空闲页面链表的每个元素，直到le指向链表头为止（实现双向链表的遍历）。使用函数le2page将初始化的page的指针由le指向Page结构体头。

比较基页面base和当前页面page的地址，如果base的地址小于page的地址，将基页面base插入到当前页面page之前，以保持链表中的页面按照地址的升序排列，之后跳出循环；

如果基页面base的地址不小于当前页面page的地址，并且当前页面是链表中的最后一个页面，将基页面base添加到链表的末尾，之后跳出循环。

之后开始维护链表：

获取基页面base的前一个链表元素，检查前一个链表元素是否等于链表头，如果不是，将前一个链表元素转换为struct Page类型的指针p。检查前一个页面p和基页面base是否相邻（地址连续），如果相

邻，将前一个页面的property增加对应大小，同时清除基页面base的特殊属性，从链表中删除基页面，并将基页面指向前一个页面，以便合并它们，前一个页面合并的检查结束；

获取基页面base的后一个链表元素，检查后一个链表元素是否等于链表头，如果不是，重复上述操作，完成后一个页面的检查。

### 对于first-fit算法改进问题：

First-Fit算法是内存分配算法中的一种简单策略，它选择第一个能容纳请求大小的可用块来分配内存。虽然First-Fit具有简单性和速度的优点，但它也有一些缺点，其中一些可能导致碎片问题：

1. 外部碎片问题： First-Fit容易导致外部碎片，即分散在已分配块之间的小块未使用内存。这些小块可能会限制大块内存分配的可能性，尤其是在内存中分配大对象时。
2. 不均匀分布： 如果较小的块在较大的块之前被分配，那么它们可能导致后续较大的分配请求无法满足。

虽然First Fit的简单性对于某些应用程序可能是足够的，但对于具有更高性能和内存利用率要求的系统，可能需要采用更高级的内存分配策略。以下是一些改进First-Fit算法的方法：

1. Next-Fit： Next-Fit是First Fit的改进版本，它从上一次分配结束的地方开始搜索可用块，而不是总是从内存的开始处开始。这可以减少外部碎片，但仍然可能存在问题。
2. Best-Fit： Best-Fit选择大小与请求最接近的块来分配内存。尽管Best-Fit可以减少外部碎片，但它会引入更多的搜索开销，因为需要找到最佳匹配。
3. Worst-Fit： Worst-Fit选择最大的可用块来分配内存。虽然它可以减少外部碎片，但在实际使用中通常表现不佳。
4. 分区和合并： 一种改进方法是定期合并内存中的碎片块，以创建更大的连续块。这可以通过内存紧凑算法来实现，但可能会引入一些性能开销。
5. 分级分配： 通过将内存划分为多个不同大小的块，可以根据请求的大小选择最合适的分区进行分配。这可以提高内存利用率和性能。
6. 动态调整策略： 一些系统采用动态调整的策略，根据当前内存使用情况选择最适合的分配策略，以在不同情况下获得最佳性能。

改进First Fit算法的方法因系统的具体需求和应用场景而异。在实际应用中，通常需要综合考虑内存分配的性能、内存利用率和复杂性，选择适合特定情况的分配策略。不同系统可能会采用不同的策略来平衡这些因素。

## 练习2:实现 Best-Fit 连续物理内存分配算法

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。请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放，并回答如下问题：

- 你的 Best-Fit 算法是否有进一步的改进空间？

清空当前页框的标志和属性信息，并将页框的引用计数设置为0

```
1 p->flags=0;
2 p->property=0;
3 set_page_ref(p,0);
```

当 $base < page$ 时，找到第一个大于 $base$ 的页，将 $base$ 插入到它前面，并退出循环；当 $list\_next(le) == \&free\_list$ 时，若已经到达链表结尾，将 $base$ 插入到链表尾部

```
1 if(base<page){
2     list_add_before(le, &(base->page_link));
3     break;
4
5 }
6
7 else if(list_next(le) == &free_list){
8     list_add_after(le, &(base->page_link));
9 }
```

best-fit实现：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

```
1 static struct Page *
2 best_fit_alloc_pages(size_t n) {
3     assert(n > 0);
4     if (n > nr_free) {
5         return NULL;
6     }
7     struct Page *page = NULL;
8     list_entry_t *le = &free_list;
9     unsigned int min_size = nr_free + 1;
10
11     while ((le = list_next(le)) != &free_list) {
12         struct Page *p = le2page(le, page_link);
13         if ((p->property >= n) && (p->property < min_size)) {
14             page = p;
15             min_size = p->property;
16             //min_fixed_page_location = p;
17             //break;
18         }
19     }
```

```

19     }
20
21     if (page != NULL) {
22         list_entry_t* prev = list_prev(&(page->page_link));
23         list_del(&(page->page_link));
24         if (page->property > n) {
25             struct Page *p = page + n;
26             p->property = page->property - n;
27             SetPageProperty(p);
28             list_add(prev, &(p->page_link));
29         }
30         nr_free -= n;
31         ClearPageProperty(page);
32     }
33     return page;
34 }

```

设置当前页块的属性为释放的页块数、并将当前页块标记为已分配状态、最后增加nr\_free的值

```

1     base->property = n;
2     SetPageProperty(base);
3     nr_free += n;

```

编写代码，要求实现以下功能：

- 1、判断前面的空闲页块是否与当前页块是连续的，如果是连续的，则将当前页块合并到前面的空闲页块中
- 2、首先更新前一个空闲页块的大小，加上当前页块的大小
- 3、清除当前页块的属性标记，表示不再是空闲页块
- 4、从链表中删除当前页块
- 5、将指针指向前一个空闲页块，以便继续检查合并后的连续空闲页块

```

1     list_entry_t* le = list_prev(&(base->page_link));
2     if (le != &free_list) {
3         p = le2page(le, page_link);
4
5         if (p + p->property == base) {
6             p->property += base->property;
7             ClearPageProperty(base);
8             list_del(&(base->page_link));
9             base = p;

```



```

10     }
11 }
12
13 le = list_next(&(base->page_link));
14 if (le != &free_list) {
15     p = le2page(le, page_link);
16     if (base + base->property == p) {
17         base->property += p->property;
18         ClearPageProperty(p);
19         list_del(&(p->page_link));
20     }
21 }

```

## 对best fit算法的改进问题：

此算法目前的实现是遍历整个空闲链表，找到满足需求的空闲页框中最小的一个。但是这种方式效率不高，因为需要遍历整个链表。

1. 一种改进的方式是使用二叉搜索树来存储空闲页框的信息。每个节点保存的是一个空闲页框的大小和地址。这样可以快速找到满足需求的最小大小的空闲页框。每次分配或释放页面时，都可以在二叉搜索树上进行查找和更新。
2. 另一种改进的方式是使用分配器索引。将空闲页框按照大小分成多个链表，并且维护一个索引数组，其中索引数组的每个元素指向一个大小范围内的链表的头节点。这样可以快速定位到满足需求的最小大小的链表，然后再在链表上进行查找和更新。
3. 优化页框拆分：在当前的实现中，如果找到的页面的property大于n，会将其拆分成两个页面。可以考虑在拆分时避免产生过小的页面，以减少碎片化问题。例如，可以设置一个最小拆分阈值，只有当页面的property大于等于该阈值时才进行拆分。

通过 `make grade` 测试：

