第二次实验报告

练习一：理解first-fit 连续物理内存分配算法

首先我们对 default_pmm.c 文件中出现的 default_init，default_init_memmap ，default_alloc_pages ， default_free_pages 等相关函数函数进行逐个分析。

default_init函数--初始化函数

函数内容如下：

代码块

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

通过 list_init 函数将空闲链表 free_list 初始化，接着将空闲页的数量 nr_free 设为0，从而完成了对空闲链表管理结构的初始化

default_init函数--初始化函数

函数内容如下：

代码块

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

通过 list_init 函数将空闲链表 free_list 初始化，接着将空闲页的数量 nr_free 设为0，从而完成了对空闲链表管理结构的初始化

default_init_memmap函数--初始化内存映射

函数内容如下：

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

此函数的作用是将一段连续的物理内存进行初始化并且加入到空闲链表当中。

这个函数将结构体指针 base 和内存块大小 n 作为变量传入， base 指针指向将要被初始化的物理内存块的起始地址。

assert 函数对 n 进行检查（大小不能为负），接着通过 for 循环将这些内存块的标志位和属性值全部清除。接着更改该内存区域头部的块参数，包括属性值和块大小等。

最后通过一些 if、else 语句完成空闲链表的插入：如果链表为空，那么直接进行插入；如果不为空，则根据地址大小比较，找到合适的地方插入

通过上面的两个函数，我们已经完成了对连续的未被引用和分配的空闲物理内存页面的初始化，接下来我们需要通过 default_alloc_pages 函数完成页面的分配

default_alloc_pages函数--分配页面

代码块

```
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;
}
```

首先依旧通过 assert 函数是检验输入的合法性，并通过一条 if 语句对空闲页面的大小进行判断，如果需要分配的内存空间大小大于空闲链表中的空闲页面，则分配失败；反之，说明满足分配条件。
接着，将 page 指针设为 NULL ，并用指针 le 指向空闲链表，方便后面通过 while 循环找到合适的页面。
接着进入到 while 循环内部，循环条件表达式完成了对链表的遍历并限定了退出条件，确保不会一直循环下去（这是个双向链表）。第一行代码 struct Page *p = le2page(le, page_link);的作用是让 p 指针指向 page 的头部而不是其内部的链表节点。 if 语句对该 page 块大小与调用需要的内存大小进行比较，从而找到合适的 page 进行分配。
后面的 if 语句则是将分配后剩下的块（如果有）重新链入到链表中，并修改参数，完成分配后的管理。首先通过list_entry_t* prev = list_prev(&(page->page_link));定位到前一个节点，并用list_del函数删除被分配的 page 块，接着再用一个 if 语句对该 page 块超过分配需求的部分进行拆分，再重新为剩余的 page 设置参数和标志，最终用 list_add 函数将其重新放入链表，等待后续的分配。

完成了页面的分配后，我们已经完成了物理内存分配的大部分内容，但是，还需要考虑进程在使用完这些内存空间后，他们需要重新回到链表，等待其他进程的调用，这就是内存分配的最后一个过程--释

放页面!

default_free_pages函数--释放页面

代码块

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

首先，通过for (; p != base + n; p ++) { assert(!PageReserved(p) && !PageProperty(p)); p->flags = 0; set_page_ref(p, 0); }对被释放的 page 进行参数设置，清除其所有的标志，assert 语句确保被释放的 page 既不是保留状态也不是空闲状态，将这些操作都完成后，为这个 page 块设置其属性值并重新加入到链表中。后面的两条 if 语句则是合并空闲块的关键步骤：
首先让指针 le 定位到该 page 块在链表中的前一个块的块头，然后用if (p + p->property == base)检查两个块的物理地址是否相邻（链表中相邻不一定意味着物理地址中也是相邻的），在确认相邻后，将两个块合并为一块，并清除后面块的属性和标志，更新前面块的属性和标志，删去二者之间的链表指针。这样，就实现了向前合并的操作，向后合并也是此理，这里不再进行分析。
到此，我们对first-fit 连续物理内存分配算法中关键函数的分析就完成了，从上面的分析不难看出，程序在进行物理内存分配的过程大致可以分为三个阶段，分别是：**初始化阶段**、**分配阶段**以及**释放阶段**。
在初始阶段，系统启动时调用 default_init 函数初始化空闲链表， 并通过 default_init_memmap 函数将可用的物理内存区域初始化为空闲块。
在分配阶段，通过 default_alloc_pages 函数扫描空闲链表，找到首个满足大小的空闲块进行分配或分割，并更新空闲页面计数。
在释放阶段，由 default_free_pages 函数将释放的块重新链入空闲链表，并通过前后合并操作将相邻空闲块组合成更大的连续空间。这就是 first-fit 算法进行连续物理内存分配的过程！

算法改进

初始化内存映射的改进

修改后的代码：

代码块

```
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 = 0;
        set_page_ref(p, 0);
```

```
10          if (p != base) {
11              p->property = 0;
12          }
13      }
14
15      base->property = n;
16      SetPageProperty(base);
17      nr_free += n;
18      list_entry_t *le = &free_list;
19      while ((le = list_next(le)) != &free_list) {
20          p = le2page(le, page_link);
21          if (base < p) {
22              break;
23          }
24      }
25      list_add_before(le, &(base->page_link));
26  }
```

源代码中通过for循环对每个 page 的属性值赋值为0，退出循环后再为块头赋值，这样存在冗余操作；修改后在循环内通过条件判断，只有非首页才设置property为0，逻辑更清晰。

同时，链表插入操作也存在问题，源代码中需要单独处理空链表情况，且在遍历时需要特殊处理到达链表末尾的情况，因此使用了两种不同的插入函数list_add_before 和 list_add ；修改后可以对空链表和非空链表使用相同的处理逻辑，并利用list_add_before函数的特性：

如果 le 指向链表头，list_add_before 相当于在链表开头插入

如果 le 指向某个节点，list_add_before 在该节点前插入

如果 le 指向链表头（遍历完整个链表），list_add_before 在链表末尾插入

这样减少了条件分支，代码可读性更高。

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

一、代码内容

代码块

```
1
2   #include <pmm.h>
3   #include <list.h>
4   #include <string.h>
5   #include <best_fit_pmm.h>
6   #include <stdio.h>
7
```

```
 8  /* In the first fit algorithm, the allocator keeps a list of free blocks
    (known as the free list) and,
 9     on receiving a request for memory, scans along the list for the first block
    that is large enough to
10     satisfy the request. If the chosen block is significantly larger than that
    requested, then it is
11     usually split, and the remainder added to the list as another free block.
12     Please see Page 196~198, Section 8.2 of Yan Wei Min's chinese book "Data
    Structure -- C programming language"
13  */
14  // LAB2 EXERCISE 1: YOUR CODE
15  // you should rewrite functions:
    default_init,default_init_memmap,default_alloc_pages, default_free_pages.
16  /*
17   * Details of FFMA
18   * (1) Prepare: In order to implement the First-Fit Mem Alloc (FFMA), we
    should manage the free mem block use some list.
19   *              The struct free_area_t is used for the management of free mem
    blocks. At first you should
20   *              be familiar to the struct list in list.h. struct list is a
    simple doubly linked list implementation.
21   *              You should know howto USE: list_init,
    list_add(list_add_after), list_add_before, list_del, list_next, list_prev
22   *              Another tricky method is to transform a general list struct to
    a special struct (such as struct page):
23   *              you can find some MACRO: le2page (in memlayout.h), (in future
    labs: le2vma (in vmm.h), le2proc (in proc.h),etc.)
24   * (2) default_init: you can reuse the  demo default_init fun to init the
    free_list and set nr_free to 0.
25   *              free_list is used to record the free mem blocks. nr_free is
    the total number for free mem blocks.
26   * (3) default_init_memmap:  CALL GRAPH: kern_init --> pmm_init-->page_init--
    >init_memmap--> pmm_manager->init_memmap
27   *              This fun is used to init a free block (with parameter:
    addr_base, page_number).
28   *              First you should init each page (in memlayout.h) in this free
    block, include:
29   *                  p->flags should be set bit PG_property (means this page is
    valid. In pmm_init fun (in pmm.c),
30   *                  the bit PG_reserved is setted in p->flags)
31   *                  if this page  is free and is not the first page of free
    block, p->property should be set to 0.
32   *                  if this page  is free and is the first page of free block,
    p->property should be set to total num of block.
33   *                  p->ref should be 0, because now p is free and no reference.
34   *                  We can use p->page_link to link this page to free_list,
    (such as: list_add_before(&free_list, &(p->page_link)); )
```

```
 *              Finally, we should sum the number of free mem block: nr_free+=n
 * (4) default_alloc_pages: search find a first free block (block size >=n) in free list and reszie the free block, return the addr
 *              of malloced block.
 *              (4.1) So you should search freelist like this:
 *                       list_entry_t le = &free_list;
 *                       while((le=list_next(le)) != &free_list) {
 *                       ....
 *                 (4.1.1) In while loop, get the struct page and check the p->property (record the num of free block) >=n?
 *                       struct Page *p = le2page(le, page_link);
 *                       if(p->property >= n){ ...
 *                 (4.1.2) If we find this p, then it' means we find a free block(block size >=n), and the first n pages can be malloced.
 *                      Some flag bits of this page should be setted: PG_reserved =1, PG_property =0
 *                      unlink the pages from free_list
 *                      (4.1.2.1) If (p->property >n), we should re-caluclate number of the the rest of this free block,
 *                           (such as: le2page(le,page_link))->property = p->property - n;)
 *                 (4.1.3)  re-caluclate nr_free (number of the the rest of all free block)
 *                 (4.1.4)  return p
 *               (4.2) If we can not find a free block (block size >=n), then return NULL
 * (5) default_free_pages: relink the pages into  free list, maybe merge small free blocks into big free blocks.
 *               (5.1) according the base addr of withdrawed blocks, search free list, find the correct position
 *                      (from low to high addr), and insert the pages. (may use list_next, le2page, list_add_before)
 *               (5.2) reset the fields of pages, such as p->ref, p->flags (PageProperty)
 *               (5.3) try to merge low addr or high addr blocks. Notice: should change some pages's p->property correctly.
 */
static free_area_t free_area;

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

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

```
69
70  static void
71  best_fit_init_memmap(struct Page *base, size_t n) {
72      assert(n > 0);
73      struct Page *p = base;
74      for (; p != base + n; p ++) {
75          assert(PageReserved(p));
76          /*LAB2 EXERCISE 2: YOUR CODE*/
77          // 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
78          p->flags = 0;
79          set_page_ref(p, 0);
80          p->property = 0;
81
82      }
83      base->property = n;
84      SetPageProperty(base);
85      nr_free += n;
86      if (list_empty(&free_list)) {
87          list_add(&free_list, &(base->page_link));
88      } else {
89          list_entry_t* le = &free_list;
90
91          while ((le = list_next(le)) != &free_list) {
92              struct Page* page = le2page(le, page_link);
93              /*LAB2 EXERCISE 2: YOUR CODE*/
94              // 编写代码
95              // 1、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
96              // 2、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
97              if (base < page) {
98                  list_add_before(le, &(base->page_link));
99                  break;
100             } else if (list_next(le) == &free_list) {
101                 list_add(le, &(base->page_link));
102                 break;
103             }
104         }
105     }
106 }
107
108 static struct Page *
109 best_fit_alloc_pages(size_t n) {
110     assert(n > 0);
111     if (n > nr_free) {
112         return NULL;
113     }
```

```
114         struct Page *page = NULL;
115         list_entry_t *le = &free_list;
116         size_t min_size = nr_free + 1;
117         /*LAB2 EXERCISE 2: YOUR CODE*/
118         // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
119         // 遍历空闲链表，查找满足需求的空闲页框
120         // 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
121         while ((le = list_next(le)) != &free_list) {
122             struct Page *p = le2page(le, page_link);
123             if (p->property >= n && p->property < min_size) {
124                 min_size = p->property;
125                 page = p;
126             }
127         }
128 
129         if (page != NULL) {
130             list_entry_t* prev = list_prev(&(page->page_link));
131             list_del(&(page->page_link));
132             if (page->property > n) {
133                 struct Page *p = page + n;
134                 p->property = page->property - n;
135                 SetPageProperty(p);
136                 list_add(prev, &(p->page_link));
137             }
138             nr_free -= n;
139             ClearPageProperty(page);
140         }
141         return page;
142     }
143 
144     static void
145     best_fit_free_pages(struct Page *base, size_t n) {
146         assert(n > 0);
147         struct Page *p = base;
148         for (; p != base + n; p ++) {
149             assert(!PageReserved(p) && !PageProperty(p));
150             p->flags = 0;
151             set_page_ref(p, 0);
152         }
153         /*LAB2 EXERCISE 2: YOUR CODE*/
154         // 编写代码
155         // 具体来说就是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
156         base->property = n;
157         SetPageProperty(base);
158         nr_free += n;
159 
```

```
160         if (list_empty(&free_list)) {
161             list_add(&free_list, &(base->page_link));
162         } else {
163             list_entry_t* le = &free_list;
164             while ((le = list_next(le)) != &free_list) {
165                 struct Page* page = le2page(le, page_link);
166                 if (base < page) {
167                     list_add_before(le, &(base->page_link));
168                     break;
169                 } else if (list_next(le) == &free_list) {
170                     list_add(le, &(base->page_link));
171                 }
172             }
173         }
174 
175         list_entry_t* le = list_prev(&(base->page_link));
176         if (le != &free_list) {
177             p = le2page(le, page_link);
178             /*LAB2 EXERCISE 2: YOUR CODE*/
179             // 编写代码
180             // 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
181             // 2、首先更新前一个空闲页块的大小，加上当前页块的大小
182             // 3、清除当前页块的属性标记，表示不再是空闲页块
183             // 4、从链表中删除当前页块
184             // 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
185             if (p + p->property == base) {
186                 p->property += base->property;
187                 ClearPageProperty(base);
188                 list_del(&(base->page_link));
189                 base = p;
190             }
191         }
192 
193         le = list_next(&(base->page_link));
194         if (le != &free_list) {
195             p = le2page(le, page_link);
196             if (base + base->property == p) {
197                 base->property += p->property;
198                 ClearPageProperty(p);
199                 list_del(&(p->page_link));
200             }
201         }
202     }
203 
204     static size_t
205     best_fit_nr_free_pages(void) {
```

```
206         return nr_free;
207     }
208
209     static void
210     basic_check(void) {
211         struct Page *p0, *p1, *p2;
212         p0 = p1 = p2 = NULL;
213         assert((p0 = alloc_page()) != NULL);
214         assert((p1 = alloc_page()) != NULL);
215         assert((p2 = alloc_page()) != NULL);
216
217         assert(p0 != p1 && p0 != p2 && p1 != p2);
218         assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);
219
220         assert(page2pa(p0) < npage * PGSIZE);
221         assert(page2pa(p1) < npage * PGSIZE);
222         assert(page2pa(p2) < npage * PGSIZE);
223
224         list_entry_t free_list_store = free_list;
225         list_init(&free_list);
226         assert(list_empty(&free_list));
227
228         unsigned int nr_free_store = nr_free;
229         nr_free = 0;
230
231         assert(alloc_page() == NULL);
232
233         free_page(p0);
234         free_page(p1);
235         free_page(p2);
236         assert(nr_free == 3);
237
238         assert((p0 = alloc_page()) != NULL);
239         assert((p1 = alloc_page()) != NULL);
240         assert((p2 = alloc_page()) != NULL);
241
242         assert(alloc_page() == NULL);
243
244         free_page(p0);
245         assert(!list_empty(&free_list));
246
247         struct Page *p;
248         assert((p = alloc_page()) == p0);
249         assert(alloc_page() == NULL);
250
251         assert(nr_free == 0);
252         free_list = free_list_store;
```

```
253         nr_free = nr_free_store;
254 
255         free_page(p);
256         free_page(p1);
257         free_page(p2);
258     }
259 
260     // LAB2: below code is used to check the best fit allocation algorithm (your
    EXERCISE 1)
261     // NOTICE: You SHOULD NOT CHANGE basic_check, default_check functions!
262     static void
263     best_fit_check(void) {
264         int score = 0 ,sumscore = 6;
265         int count = 0, total = 0;
266         list_entry_t *le = &free_list;
267         while ((le = list_next(le)) != &free_list) {
268             struct Page *p = le2page(le, page_link);
269             assert(PageProperty(p));
270             count ++, total += p->property;
271         }
272         assert(total == nr_free_pages());
273 
274         basic_check();
275 
276         #ifdef ucore_test
277         score += 1;
278         cprintf("grading: %d / %d points\n",score, sumscore);
279         #endif
280         struct Page *p0 = alloc_pages(5), *p1, *p2;
281         assert(p0 != NULL);
282         assert(!PageProperty(p0));
283 
284         #ifdef ucore_test
285         score += 1;
286         cprintf("grading: %d / %d points\n",score, sumscore);
287         #endif
288         list_entry_t free_list_store = free_list;
289         list_init(&free_list);
290         assert(list_empty(&free_list));
291         assert(alloc_page() == NULL);
292 
293         #ifdef ucore_test
294         score += 1;
295         cprintf("grading: %d / %d points\n",score, sumscore);
296         #endif
297         unsigned int nr_free_store = nr_free;
298         nr_free = 0;
```

```
299
300     // * - - * -
301     free_pages(p0 + 1, 2);
302     free_pages(p0 + 4, 1);
303     assert(alloc_pages(4) == NULL);
304     assert(PageProperty(p0 + 1) && p0[1].property == 2);
305     // * - - * *
306     assert((p1 = alloc_pages(1)) != NULL);
307     assert(alloc_pages(2) != NULL);      // best fit feature
308     assert(p0 + 4 == p1);
309
310     #ifdef ucore_test
311     score += 1;
312     cprintf("grading: %d / %d points\n",score, sumscore);
313     #endif
314     p2 = p0 + 1;
315     free_pages(p0, 5);
316     assert((p0 = alloc_pages(5)) != NULL);
317     assert(alloc_page() == NULL);
318
319     #ifdef ucore_test
320     score += 1;
321     cprintf("grading: %d / %d points\n",score, sumscore);
322     #endif
323     assert(nr_free == 0);
324     nr_free = nr_free_store;
325
326     free_list = free_list_store;
327     free_pages(p0, 5);
328
329     le = &free_list;
330     while ((le = list_next(le)) != &free_list) {
331         struct Page *p = le2page(le, page_link);
332         count --, total -= p->property;
333     }
334     assert(count == 0);
335     assert(total == 0);
336     #ifdef ucore_test
337     score += 1;
338     cprintf("grading: %d / %d points\n",score, sumscore);
339     #endif
340 }
341
342 const struct pmm_manager best_fit_pmm_manager = {
343     .name = "best_fit_pmm_manager",
344     .init = best_fit_init,
345     .init_memmap = best_fit_init_memmap,
```

```
346       .alloc_pages = best_fit_alloc_pages,
347       .free_pages = best_fit_free_pages,
348       .nr_free_pages = best_fit_nr_free_pages,
349       .check = best_fit_check,
350   };
351
```

二、算法分析

1.best_fit_init_memmap 函数

best_fit_init_memmap 函数负责初始化一段连续的物理内存页面，将其建立为一个可管理的空闲内存块。当系统启动时，物理内存管理器通过此函数将可用的内存区域注册到管理系统中。函数首先对传入的每个页面进行合法性检查，确保它们确实处于保留状态，然后逐一初始化每个页面的元数据：清除所有状态标志、将引用计数归零，并将非头页面的property属性设置为0。接着，函数将这段连续页面的第一个页面标记为空闲块的头页面，设置其property属性为整个块的大小，并为其添加PG_property标志以表明这是一个空闲块的起始页面。最后，函数将这个新初始化的空闲块按照物理地址升序插入到空闲链表中，确保链表始终保持有序排列，这种有序性为后续的内存合并操作提供了便利，同时更新系统总空闲页面计数。

2. best_fit_alloc_pages 函数

best_fit_alloc_pages函数是Best-Fit算法的核心实现，负责从空闲内存中分配指定数量的连续物理页面。函数开始时首先进行基本的参数验证，检查请求的页面数量是否有效以及系统中是否有足够的空闲页面。随后进入Best-Fit的核心搜索阶段：函数遍历整个空闲链表，对于每个空闲块，检查其大小是否满足分配需求，并在所有满足条件的块中记录大小最接近需求的那个块，这正是Best-Fit与First-Fit的关键区别——Best-Fit追求的是最小浪费而非首次匹配。找到最佳候选块后，函数将其从空闲链表中移除，并进行块分割处理：如果选中的块比实际需求大，则将剩余部分创建为新的空闲块，设置适当的属性后重新插入链表中原位置，这样可以有效减少内部碎片。最后，函数更新系统空闲页面计数，清除分配页面的属性标志，并返回分配块的首页面指针。

3.best_fit_free_pages 函数

best_fit_free_pages函数负责释放之前分配的物理内存页面，将其重新纳入空闲内存管理系统。函数执行时首先验证释放参数的合法性，然后对要释放的每个页面进行状态重置：清除所有标志位、将引用计数归零，确保页面恢复到可重新分配的状态。接着，函数将这些释放的页面组织成一个新的空闲块，将第一个页面标记为块头，设置相应的property属性和PG_property标志，并按照物理地址顺序将其插入空闲链表。最重要的步骤是内存合并操作：函数检查新插入块与其前后相邻块在物理地址上是否连续，如果发现与前一个块相邻，则进行向前合并，将当前块合并到前一个块中，更新块大小并调整链表；同样地，如果与后一个块相邻，则进行向后合并。

4.块合并机制

块合并机制是内存管理系统中减少外部碎片的关键策略，在Best-Fit算法的释放过程中发挥着重要作用。该机制基于物理地址的连续性检查来实现：当释放一个内存块时，系统会检查该块是否与已有的空闲块在物理地址上相邻。向前合并检查是通过获取当前块在链表中的前驱节点，计算前驱块的结束地址是否正好等于当前块的起始地址，如果满足这一条件，说明两个块在物理内存中是连续的，系统会将当前块合并到前驱块中，增大前驱块的property值，同时从链表中移除当前块的节点。向后合并采用类似的逻辑，检查当前块的结束地址是否等于后继块的起始地址。这种双向合并确保了无论新释放的块与哪个方向的空闲块相邻，都能被正确合并，从而最大限度地整合零散的内存空间，形成更大的连续空闲区域，显著提升内存利用效率。

三、设计思路及如何对物理内存进行分配和释放

Best-Fit内存管理算法采用了一种基于"最小浪费原则"的设计理念，目标是在所有可用空闲块中选择大小最接近请求值的块进行分配，从而最大限度地减少内存内部碎片。

算法执行流程大致分为三个阶段：在系统初始化阶段，首先通过pmm_init注册Best-Fit管理器，接着page_init从设备树获取物理内存范围并划分可用区域，最后best_fit_init_memmap对每个空闲页面进行元数据初始化，设置正确的标志位和引用计数，并将整个块按地址顺序插入空闲链表。

当应用程序请求内存分配时，best_fit_alloc_pages开始工作，它首先验证请求的合法性并检查资源充足性，然后进入核心的Best-Fit搜索循环——遍历整个空闲链表，在所有大小满足需求的块中记录属性值最接近请求值的那个块，这一全局搜索策略正是Best-Fit与First-Fit的区别。找到最佳候选块后，算法会将其从链表中移除，如果该块比实际需求大，则执行块分割操作：将剩余部分创建为新的空闲块，设置适当的属性后重新插入链表，最后更新系统空闲页面计数并返回分配结果。

当内存释时，best_fit_free_pages首先重置每个释放页面的状态标志和引用计数，然后将这些页面组织成新的空闲块并按地址顺序插入链表。随后进行双向合并机制：算法通过精确的地址连续性检查（prev + prev->property == base判断向前合并条件，base + base->property == next判断向后合并条件），自动检测并融合物理地址相邻的空闲块，这种合并策略能有效重组碎片化内存空间，将多个小空闲块整合成更大的连续区域。Best-Fit策略虽然需要全局搜索而带来稍高的时间开销，但通过最小化内部碎片获得了优越的内存利用率；同时，有序链表结构与双向合并算法的协同工作又有效缓解了外部碎片问题，共同构成了一个既保证分配精度又控制管理开销的完整内存管理解决方案。

四、测试结果

```
lcg2@LAPTOP-0DMKOI99:/mnt/e/课程内容/操作系统/labcode/labcode/lab2$ make
+ cc kern/mm/best_fit_pmm.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img
lcg2@LAPTOP-0DMKOI99:/mnt/e/课程内容/操作系统/labcode/labcode/lab2$ make qemu

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    ____ _____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc020164a (virtual)
  edata  0xffffffffc0205018 (virtual)
  end    0xffffffffc0205078 (virtual)
```

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc020164a (virtual)
  edata  0xffffffffc0205018 (virtual)
  end    0xffffffffc0205078 (virtual)
Kernel executable memory footprint: 20KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0204000
satp physical address: 0x0000000080204000
QEMU: Terminated
lcg2@LAPTOP-0DMKOI99:/mnt/e/课程内容/操作系统/labcode/labcode/lab2$
```