Week9 Report in Class (Fri56)

11911839 聂雨荷

Q1

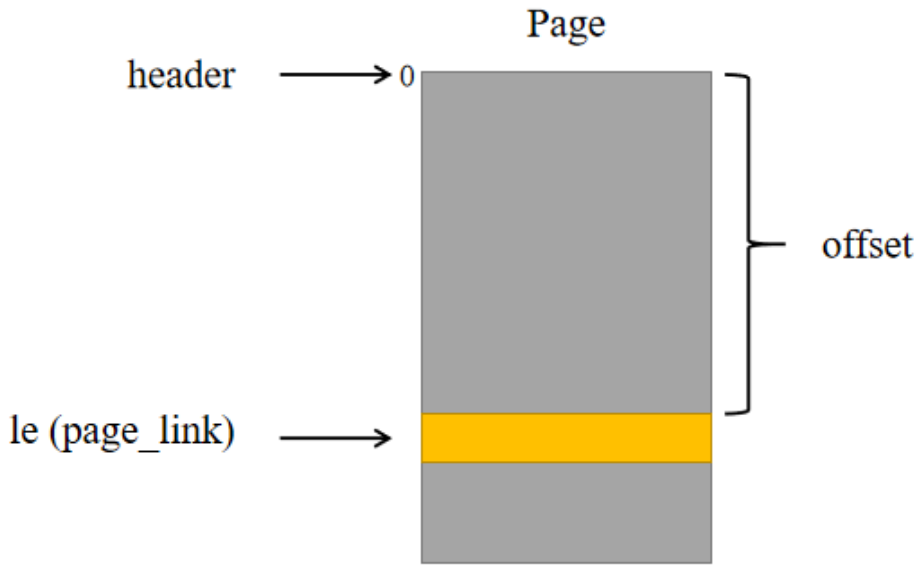
请将 `default_pmm.c` 中的85行 `le2page(le, page_link)` 宏展开，并简述 `le2page` 的工作原理（可以画图解释）

```
// convert list entry to page
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)

/* *
 * to_struct - get the struct from a ptr
 * @ptr:    a struct pointer of member
 * @type:   the type of the struct this is embedded in
 * @member: the name of the member within the struct
 * */
#define to_struct(ptr, type, member)
        \
    ((type *)((char *)(ptr) - offsetof(type, member)))

/* Return the offset of 'member' relative to the beginning
of a struct type */
#define offsetof(type, member)
         \
    ((size_t)(&((type *)0)->member))
```

The code `le2page(le, page_link)` is equal to

```
((type *)((char *)(le) - ((size_t)(&((struct Page *)0)->page_link))
```

- calculate the header of the one `Page` struct according to the the pointer of `page_link` (le)



Q2

请详细描述 `default_pmm.c` 中的 `default_alloc_pages` 和 `default_free_pages` 的功能与实现方式。

defaule_alloc_pages

`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.

1. So you should search freelist like this:

```
list_entry_t le = &free_list;
while((le=list_next(le)) != &free_list) {....}
```

   - 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){ ...}
```

   - 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

     - 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;`)

- re-caluclate `nr_free` (number of the the rest of all free block)
- return p

2. If we can not find a free block (block size >=n), then return NULL

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);

    // if all free pages are smaller than the desire spaces
    // Return NULL
    if (n > nr_free) {
        return NULL;
    }

    // start from the head of the `free_list`, fine the
first caption page
    // which has continuous space >= n
    // set this page to be the desire page
    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;
        }
    }

    // remove this page from the `free_list`
    // insert the next free caption page to the `free_list`
    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;
```

```
}
```

default_free_pages

Relink the pages into free list, maybe merge small free blocks into big free blocks.

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)
2. reset the fields of pages, such as p->ref, p->flags
3. merge low addr or high addr blocks

```
static void
default_free_pages(struct Page *base, size_t n) {

    // free n pages right next to the base page
    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;

    // insert the base page back to the `free_link`
    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));
            }
        }
    }

    // merge free blocks
```

```
    // ...
}
```