CS334 Lab9 Report

以下部分是知识点的描述，对问题的作答见第二部分

Part 1

确定物理内存可分配的区间

首先，明确 qemu 模拟出来的内存以及预先含有了 opensbi 以及 kernel两个部分，所以在分配内存之前，需要确定剩余的地址空间大小。

opensbi 占用 `0x80000000` 到 `0x80200000`

kernel 占用 `0x80200000`到 `kernel end`的空间

在本次实验中，不设置特殊的地址映射，虚拟地址被指定与物理实际地址相等，只需要使用偏移量就可以获得真实的物理地址。

OpenSBI 会扫描包括 物理内存 在内的各个外设扫描，里面包括`DRAM`的信息

将可分配区间以页为单位进行分割管理

为每一个页建立一个结构体

```
struct Page {
    int ref;                        // page frame's reference counter
//
    uint64_t flags;                 // array of flags that describe
the status of the page frame
    unsigned int property;          // the num of free block, used in
first fit pm manager
    list_entry_t page_link;         // free list link
};
```

`ref`代表该页被页表引用的次数

`flags`代表该页具有的属性，1. 是否被保留，即是否可以被分配。2. 是否是空闲页，即能够被分配。

`property`记录某连续内存块的大小

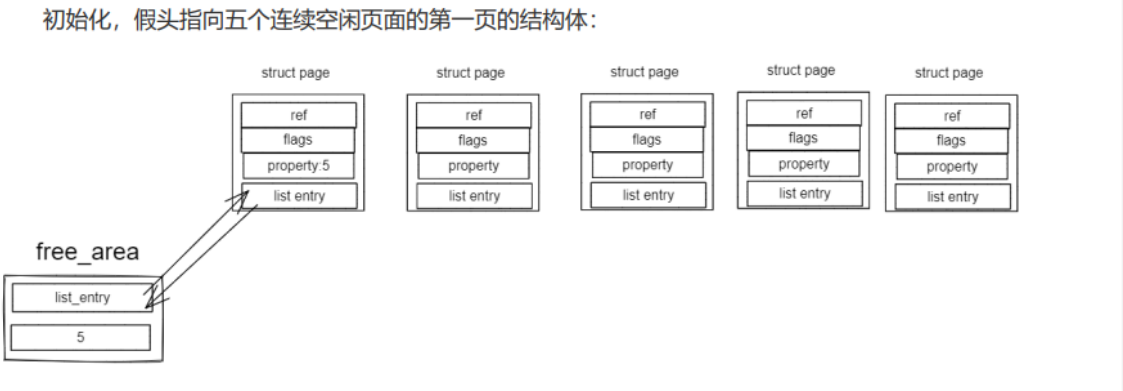
`page_link`用来链接相邻地址的其它连续内存块

注意后两个变量都只会被一块连续内存块当中的头一页使用，用来代表整块内存块的信息。

管理空闲页面

使用双向链表对空闲的连续页面进行管理。将连续的页面看做一个整体，其中这个整体的第一个页面作为“队长”，队长记录连续空闲页面空间的大小。主要是`list_entry`会指向下一个连续空闲区间的队长`list_entry`

实际中，使用一个假头来指向几个连续空闲页面的第一页的结构体



实现空闲内存区间的分配与释放

直接改变假头指向的队长`list_entry`的位置，并改变`property`的值为合适的大小即可

在实际当中，随着物理页的分配与释放，一个大的连续内存空闲块会分裂成一系列地址不连续的多个小连续内存空闲块，且每个连续内存空闲块内部的物理页是连续的。

为了便于分配和释放，定义一个`free_area_t`的数据结构

```
/* free_area_t - maintains a doubly linked list to record free
(unused) pages */
typedef struct {
list_entry_t free_list; // the list header
unsigned int nr_free; // # of free pages in this free list
} free_area_t;
```

为了管理物理内存，需要在内核中定义`page`结构体来存储记录“当前使用了哪些物理页面，哪些物理页面没有被使用”

Part 2

这部分是对问题的作答

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）

根据找到的代码

```
#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))
```

`le2page(le, page_link)`宏展开之后结果为：

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

工作原理是通过`((size_t)(&((struct Page *)0)->member))`来找到一个`page`结构体的`page_link`地址与`page`结构体相对的偏移量（将0作为这个`page_link`所在的结构体的首地址），进而可以找到`page`结构体的`ref`所在的位置

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

`default_alloc_pages`功能：在物理内存当中找到一个大小为`n`的空间并将这个空间的指针作为返回值返回

`default_free_pages`功能：释放传入的`page`指针所指向的大小为`n`的物理内存地址。

`default_alloc_pages`实现方式：通过遍历空闲列表，直到找到一块大小大于 传入的`n`的连续内存块，之后因为连续的空闲块是一个双向链表结构，此时记录队长的前一个页面`prev`，用于在之后与剩余的连续空间区域相连接。由于此时的队长已经被分配出去，所以需要将队长的`page_link`占据的空间释放掉。如果该连续的内存块空间大小大于`n`，还需要偏移对应的指针到下一个空闲的页面位置，并且将该页面对应的`property`设置为原来的`property`大小减去分配的内存，之后将`prev`与剩余连续空间块头页的`page_link`相连接。

最终修改`nr_free`为原有值减去被分配出去的内存空间大小，并且清除`page`的`property`信息，因为`page`已经被分配出去，之后再也不会被使用`property`信息。

`default_free_pages`实现方式大致与`default_alloc_pages`的实现方式相反，多了一些设置`ref`位的操作以及恢复`nt_free`的操作。

如果 当前`free_list`是空的，则直接将`page`的`page_link`插入`free_list`

如果不是空的，那么就在 `free_list` 当中找到地址符合从低到高的位置插入 `page_link`