Week6 物理内存管理

一、 实验概述

在本次实验中，我们将完成物理内存管理，并建立一个以页为管理单位的连续内存分配机制（并非分页分配）。

二、实验目的

1. 掌握物理内存管理相关的概念
2. 掌握将内存以页为单位进行管理的基本代码框架
3. 掌握连续内存分配的基本实现方式

三、实验项目整体框架概述

```
// week6
├── kern
│   ├── debug
│   ├── driver
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   ├── mm
│   │   ├── best_fit_pmm.c
│   │   ├── best_fit_pmm.h
│   │   ├── default_pmm.c //具体的内存管理函数的实现，还包括实现的检查
│   │   ├── default_pmm.h
│   │   ├── memlayout.h
│   │   ├── mmu.h
│   │   ├── pmm.c  //物理内存管理器结构的实现，定义了内存管理相关函数。
│   │   └── pmm.h
│   ├── sync
│   │   └── sync.h
│   └── trap
├── libs
│   ├── atomic.h
│   ├── defs.h
│   ├── error.h
│   ├── list.h //定义了通用双向链表结构以及相关的查找、插入等基本操作
│   ├── printfmt.c
│   ├── readline.c
│   ├── riscv.h
│   ├── sbi.c
│   ├── sbi.h
│   ├── stdarg.h
│   ├── stdio.h
│   ├── string.c
```

```
│   └── string.h
├── Makefile
└── tools
    ├── function.mk
    └── kernel.ld //通过这里获取了kernel end
```

四、实验内容

1. 确定物理内存可分配的区间
2. 将可分配区间以页为单位（大小）进行分割管理
3. 管理空闲内存区间
4. 实现空闲内存区间的分配
5. 实现已占用内存区间的释放

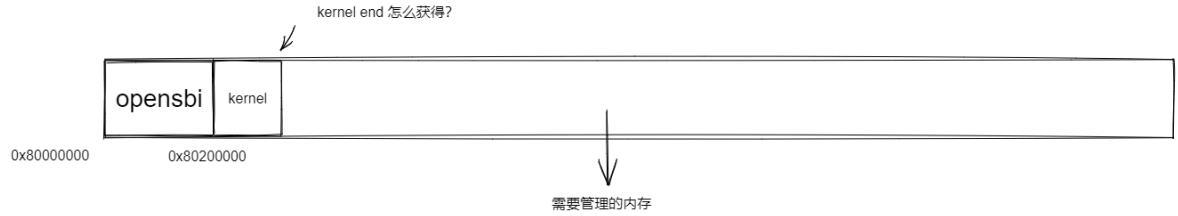
五、内存管理整体过程概述

1. 确定物理内存可分配的区间

qemu模拟出的内存中除去opensbi及kernel需要占用的两个部分，其余部分即为我们需要的内存，因此我们首先会确定这块内存所处的物理地址空间。

其中opensbi占用0x80000000到0x80200000的空间，尽管它本身并没有这么大；

kernel占用0x80200000到kernel end的空间。



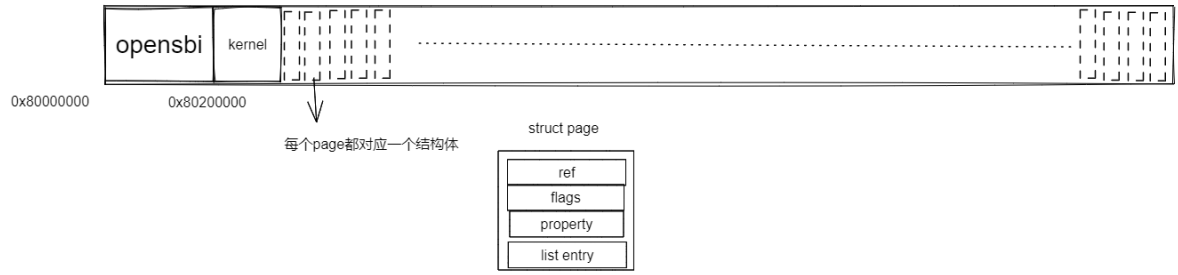
为方便管理，我们还会建立简单的虚拟地址映射，即指定一个虚拟地址与物理地址0x80200000（操作系统指定的启动地址）对齐，之后所有的虚拟地址只需进行偏移即可获得真实的物理地址。在本次实验中，为了方便理解，我们直接将物理地址与虚拟地址设置成完全相同，在后面的实验中会另外进行涉及虚拟地址映射的实验。

2. 将可分配区间以页为单位（大小）进行分割管理

在本次实验中我们并没有实现实际的分页算法（paging），但我们会将内存以页面大小为基本管理单位进行分割及分配管理。

如下图所示，我们会将需要管理的内存区间分配为等大小的若干页，页的数量由`内存区间大小÷页面大小`得到。

之后对为每一页建立一个结构体`struct page`存储页面相关的信息（如页面是否被占用等信息），并且用于管理空闲页面。



这些`page`结构体也需要占用内存空间，因此我们把它们集中放置在kernel end之后的连续几页中，以与后面可以进行的页面进行分离。

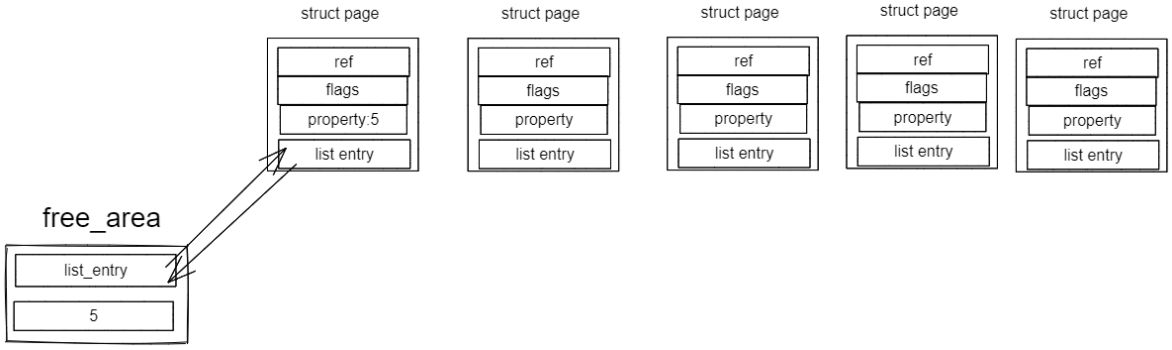
3. 管理空闲页面

我们使用双向链表对空闲的连续页面进行管理。首先每个`struct page`中有一个指针`list_entry`可以指向其他页面的`struct page`中的`list_entry`。然后我们将连续的空闲页面视为一个整体，这个整体中的第一个页面作为整体的"队长"，队长记录连续空闲页面空间的大小，并且队长的`list_entry`会指向下一个连续空闲区间的队长`list_entry`。这样我们就将所有空闲页面链接成了一个"队长"链表，以对空闲页面进行管理。

那么如何获取这个空闲空间的链表呢，我们需要在kernel空间创建一个"假的"链表头结构体`struct free_area`，假头中的`list_entry`作为头指针指向第一个空闲页面区间的队长的`list_entry`。

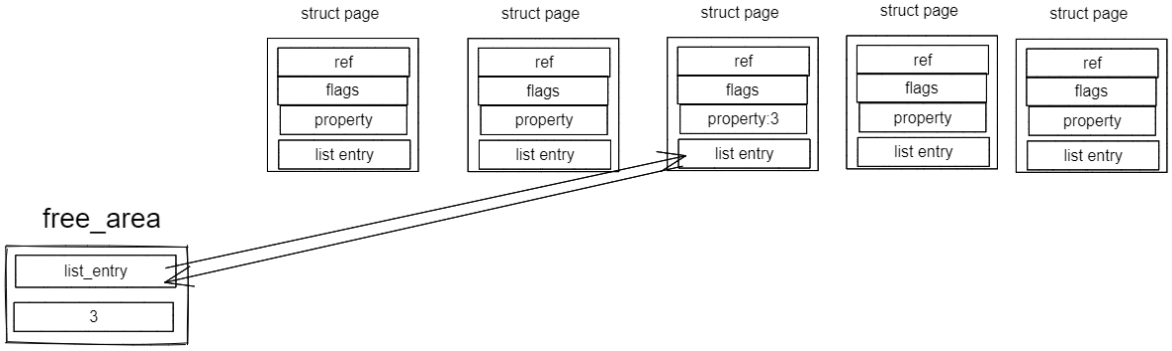
接下来我们将用一个简单5个页面的例子来介绍管理空闲页面的过程（双链表，图中省略了一些指针）。

初始化，假头指向五个连续空闲页面的第一页的结构体：

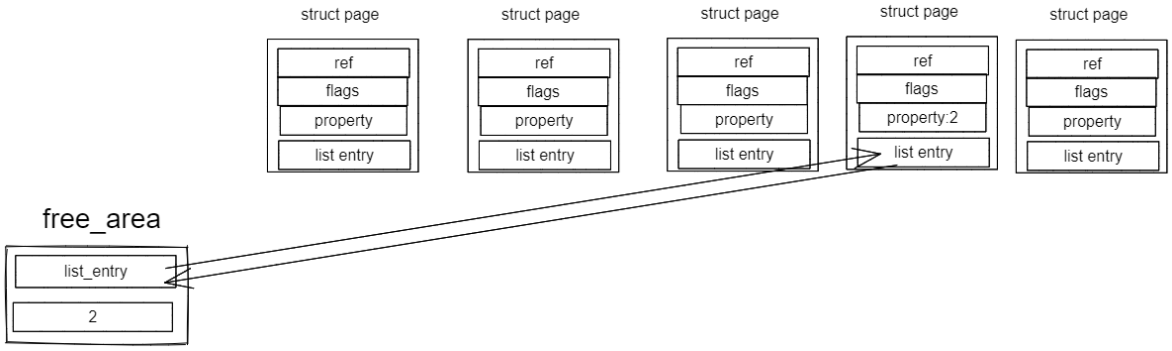


4. 实现空闲内存区间的分配

以`first fit`算法为例，我们首先分配2个页面，即通过链表找到第一个能容下2个页面的连续空闲空间进行分配，并将剩余空间插入原链表对链表进行更新：

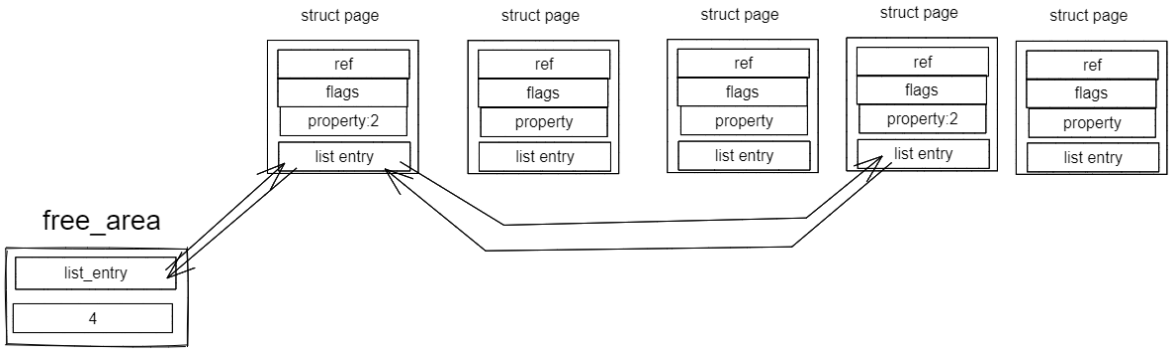


再分配1个：



5. 实现已占用内存区间的释放

此时如果回收最初分配的2个页面，则将释放的页面插入空闲空间链表以更新：



六、实验流程及相关知识点

本次实验以阅读理解代码为主，下面我们将会按照步骤给大家介绍代码实现过程。

第一步. 确定物理内存可分配空间

首先，我们需要确定需要分配的物理内存空间。

物理内存探测

操作系统怎样知道物理内存所在的那段物理地址呢？在 RISC-V 中，这个一般是由 bootloader ，即 OpenSBI 来完成的。它来完成对于包括物理内存在内的各外设的扫描，将扫描结果以 DTB(Device Tree Blob) 的格式保存在物理内存中的某个地方。随后 OpenSBI 会将其地址保存在 `a1` 寄存器中，给我们使用。

这个扫描结果描述了所有外设的信息，当中也包括 Qemu 模拟的 RISC-V 计算机中的物理内存。

> **Qemu 模拟的 RISC-V virt 计算机中的物理内存**
>
> 通过查看 `qemu-5.0.0/hw/riscv/[virt.c]` 的**virt_memmap[]**的定义，可以了解到 Qemu 模拟的 RISC-V virt 计算机的详细物理内存布局。可以看到，整个物理内存中有不少内存空洞（即含义为**unmapped**的地址空间），也有很多外设特定的地址空间，现在我们看不懂没有关系，后面会慢慢涉及到。目前只需关心最后一块含义为**DRAM**的地址空间，这就是 OS 将要管理的 128MB 的内存空间

起始地址	终止地址	含义
0x0	0x100	QEMU VIRT_DEBUG
0x100	0x1000	unmapped
0x1000	0x12000	QEMU MROM
0x12000	0x100000	unmapped
0x100000	0x101000	QEMU VIRT_TEST
0x101000	0x2000000	unmapped
0x2000000	0x2010000	QEMU VIRT_CLINT
0x2010000	0x3000000	unmapped
0x3000000	0x3010000	QEMU VIRT_PCIE_PIO
0x3010000	0xc000000	unmapped
0xc000000	0x10000000	QEMU VIRT_PLIC
0x10000000	0x10000100	QEMU VIRT_UART0
0x10000100	0x10001000	unmapped
0x10001000	0x10002000	QEMU VIRT_VIRTIO
0x10002000	0x20000000	unmapped
0x20000000	0x24000000	QEMU VIRT_FLASH
0x24000000	0x30000000	unmapped
0x30000000	0x40000000	QEMU VIRT_PCIE_ECAM
0x40000000	0x80000000	QEMU VIRT_PCIE_MMIO
0x80000000	0x88000000	DRAM 缺省 128MB，大小可配置

不过为了简单起见，我们并不打算自己去解析这个扫描结果。

由于Qemu 规定的 DRAM 物理内存的起始物理地址为 `0x80000000`（在 Qemu 中，可以使用 `-m` 指定 RAM 的大小，默认是 128MB），那么默认的 DRAM 物理内存地址范围就是 `[0x80000000,0x88000000)`。

但是，有一部分 DRAM 空间已经被占用，不能用来存别的东西了！

- 物理地址空间 `[0x80000000,0x80200000)` 被 OpenSBI 占用；
- 物理地址空间 `[0x80200000,KernelEnd)` 被内核各代码与数据段占用；
- 其实设备树扫描结果 DTB 还占用了一部分物理内存，不过由于我们不打算使用它，所以可以将它所占用的空间用来存别的东西。

于是，我们可以用来存别的东西的物理内存的物理地址范围是：`[KernelEnd, 0x88000000)`。这里的`KernelEnd`为内核代码结尾的物理地址。在`kernel.ld`中定义的`end`符号为内核代码结尾相对`BASE_ADDRESS`的地址，通过该地址我们可以得到相应的物理地址。

第二步. 将可分配区间以页为单位（大小）进行分割管理

物理内存管理应当提供这样的接口：

- 检查当前还有多少空闲的物理页，返回空闲的物理页数目
- 给出n，尝试分配n个物理页，可以返回一个起始地址和连续的物理页数目，也可能分配一些零散的物理页，返回一个连起来的链表。
- 给出起始地址和n，释放n个连续的物理页

在`kern_init()`里，我们调用一个新函数：`pmm_init()`

```
// kern/init/init.c
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init();  // init the console
    const char *message = "os is loading ...\n";
    cputs(message);
    print_kerninfo();

    idt_init();  // init interrupt descriptor table
    pmm_init();  // new!
    intr_enable();  // enable irq interrupt
    /* do nothing */
    while (1)
        ;
}

// kern/mm/pmm.c
/* pmm_init - initialize the physical memory management */
void pmm_init(void) {
    // We need to alloc/free the physical memory (granularity is 4KB or other
size).
    // So a framework of physical memory manager (struct pmm_manager)is defined
in pmm.h
    // First we should init a physical memory manager(pmm) based on the
framework.
    // Then pmm can alloc/free the physical memory.
    init_pmm_manager();

    // detect physical memory space, reserve already used memory,
    // then use pmm->init_memmap to create free page list
    page_init();

    // use pmm->check to verify the correctness of the alloc/free function in a
pmm
    check_alloc_page();
}
```

`init_pmm_manager`是初始化一个具体的默认物理内存分配器，这个分配器完成分配和回收算法的具体实现。当然，我们也可以完成多种分配算法的实现。

`page_init()`，完成了物理内存探测和页式分配算法的基础

`check_alloc_page()`是用来测试对物理内存分配功能的。

我们重点关注`page_init()`，其中，我们确定了需要分配的内存空间地址，将整个空间进行了分页（每页4096bytes），并计算得出了页的数量、空闲区的起始页。

> ## 补充
>
> 我们在代码中增加了一些功能，方便我们编程：
>
> - kern/sync/sync.h：为确保内存管理修改相关数据时不被中断打断，提供两个功能，一个是保存 sstatus寄存器中的中断使能位(SIE)信息并屏蔽中断的功能，另一个是根据保存的中断使能位信息来使能中断的功能
> - libs/list.h：定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理（以及其他内核功能）的基础。其他有类似双向链表需求的内核功能模块可直接使用 list.h 中定义的函数。
> - libs/atomic.h：定义了对一个二进制位进行读写的原子操作，确保相关操作不被中断打断。包括set_bit()设置某个二进制位的值为1, change_bit()给某个二进制位取反，test_bit()返回某个二进制位的值。
>
> `list.h`里面实现了一个简单的双向链表。虽然接口很多，但是只要对链表熟悉，不难理解。我们前面做了链表的简单实验，如果还是理解不了，可以深入学习一下链表。
>
> 看起来`list.h`里面定义的`list_entry`并没有数据域，但是，如果我们把`list_entry`作为其他结构体的成员，就可以利用C语言结构体内存连续布局的特点，从`list_entry`的地址获得它所在的上一级结构体。

接着，我们在`kern/mm/memlayout.h`中定义了可以连成链表的`Page`结构体、假链表头结构体`free_area_t`以及一系做操作的宏。

```
// libs/defs.h

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

// kern/mm/memlayout.h
/* *
 * struct Page - Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as physical address.
 * */
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
};

/* Flags describing the status of a page frame */
#define PG_reserved                 0       // if this bit=1: the Page is
reserved for kernel, cannot be used in alloc/free_pages; otherwise, this bit=0
#define PG_property                 1       // if this bit=1: the Page is the
head page of a free memory block(contains some continuous_addrress pages), and
can be used in alloc_pages; if this bit=0: if the Page is the the head page of a
free memory block, then this Page and the memory block is alloced. Or this Page
isn't the head page.
//这几个对page操作的宏用到了atomic.h的原子操作
#define SetPageReserved(page)       set_bit(PG_reserved, &((page)->flags))
#define ClearPageReserved(page)     clear_bit(PG_reserved, &((page)->flags))
#define PageReserved(page)          test_bit(PG_reserved, &((page)->flags))
#define SetPageProperty(page)       set_bit(PG_property, &((page)->flags))
#define ClearPageProperty(page)     clear_bit(PG_property, &((page)->flags))
#define PageProperty(page)          test_bit(PG_property, &((page)->flags))

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

/* 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;
```

在获得可用物理内存范围后，为了与以后的分页机制配合，系统需要建立相应的数据结构来管理以物理页（按4KB对齐，且大小为4KB的物理内存单元）为最小单位的整个物理内存，以配合后续涉及的分页管理机制。每个物理页可以用一个 Page数据结构来表示。由于一个物理页需要占用一个Page结构的空间，Page结构在设计时须尽可能小，以减少对内存的占用。Page的定义在`kern/mm/memlayout.h`中。以页为单位的物理内存分配管理的实现在`kern/default_pmm.c`。

结构Page包含了以下信息：

```
struct Page {
    int ref;        // page frame's reference counter
    uint32_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
};
```

这里看看Page数据结构的各个成员变量有何具体含义。ref表示这页被页表的引用记数（在“实现分页机制”一节会讲到）。如果这个页被页表引用了，即在某页表中有一个页表项设置了一个虚拟页到这个Page管理的物理页的映射关系，就会把Page的ref加一；反之，若页表项取消，即映射关系解除，就会把Page的ref减一。flags表示此物理页的状态标记，进一步查看`kern/mm/memlayout.h`中的定义，可以看到：

```
/* Flags describing the status of a page frame */
#define PG_reserved                 0       // the page descriptor is reserved
for kernel or unusable
#define PG_property                 1       // the member 'property' is valid
```

这表示flags目前用到了两个bit表示页目前具有的两种属性，bit 0表示此页是否被保留（reserved），如果是被保留的页，则bit 0会设置为1，且不能放到空闲页链表中，即这样的页不是空闲页，不能动态分配与释放。比如目前内核代码占用的空间就属于这样“被保留”的页。在本实验中，bit 1表示此页是否是free的，如果设置为1，表示这页是free的，可以被分配；如果设置为0，表示这页已经被分配出去了，不能被再二次分配。另外，本实验这里取的名字PG_property比较不直观，主要是我们可以设计不同的页分配算法（best fit, buddy system等），那么这个PG_property就有不同的含义了。

Page数据结构的成员变量property用来记录某连续内存空闲块的大小（即地址连续的空闲页的个数）。这里需要注意的是用到此成员变量的这个Page比较特殊，是这个连续内存空闲块地址最小的一页（即头一页， Head Page）。连续内存空闲块利用这个页的成员变量property来记录在此块内的空闲页的个数。这里取的名字property也不是很直观，原因与上面类似，在不同的页分配算法中，property有不同的含义。

Page数据结构的成员变量page_link是便于把多个连续内存空闲块链接在一起的双向链表指针。这里需要注意的是用到此成员变量的这个Page比较特殊，是这个连续内存空闲块地址最小的一页（即头一页，Head Page）。连续内存空闲块利用这个页的成员变量page_link来链接比它地址小和大的其他连续内存空闲块。

free_area_t结构体

在初始情况下，也许这个物理内存的空闲物理页都是连续的，这样就形成了一个大的连续内存空闲块。但随着物理页的分配与释放，这个大的连续内存空闲块会分裂为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。那么为了有效地管理这些小连续内存空闲块，所有的连续内存空闲块可用一个双向链表管理起来。为了便于分配和释放，定义了一个free_area_t数据结构，包含了一个list_entry结构的双向链表指针和记录当前空闲页的个数的无符号整型变量nr_free。其中的链表指针指向了空闲的物理页。

```
/* 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;
```

初始化Page结构体

为了管理物理内存，我们需要在内核里定义一些数据结构，来存储“当前使用了哪些物理页面，哪些物理页面没被使用”这样的信息，使用的是Page结构体。我们将一些Page结构体在内存里排列在`kernel_end`后面，这要占用一些内存。而摆放这些Page结构体的物理页面，以及内核占用的物理页面，之后都无法再使用了。实验代码`/kern/mm/default_pmm.c`中的default_init_memmap()函数对这些page结构体进行了初始化并将链表进行了链接。

> ### C语言中的“面向对象”
>
> 我们用`page_init()`函数给这些管理物理内存的结构体做初始化。`page_init()`的代码里，我们调用了一个函数`init_memmap()`，这和我们的另一个结构体`pmm_manager`有关。虽然C语言基本上不支持面向对象，但我们可以用类似面向对象的思路，把“物理内存管理”的功能集中给一个结构体。我们甚至可以让函数指针作为结构体的成员，强行在C语言里支持了“成员函数”。可以看

到，我们调用的`init_memmap()`实际上又调用了`pmm_manager`的一个"成员函数"。

`pmm_manager`提供了各种接口：分配页面，释放页面，查看当前空闲页面数。但是我们好像始终没看见`pmm_manager`内部对这些接口的实现，那些接口只是作为函数指针，作为`pmm_manager`的一部分，我们需要把那些函数指针变量赋值为真正的函数名称。在这里面我们把pmm_manager的指针赋值成`&default_pmm_manager`。

```
// init_pmm_manager - initialize a pmm_manager instance
static void init_pmm_manager(void) {
 pmm_manager = &default_pmm_manager;
 cprintf("memory management: %s\n", pmm_manager->name);
 pmm_manager->init();
}
```

初始化free_area_t结构体

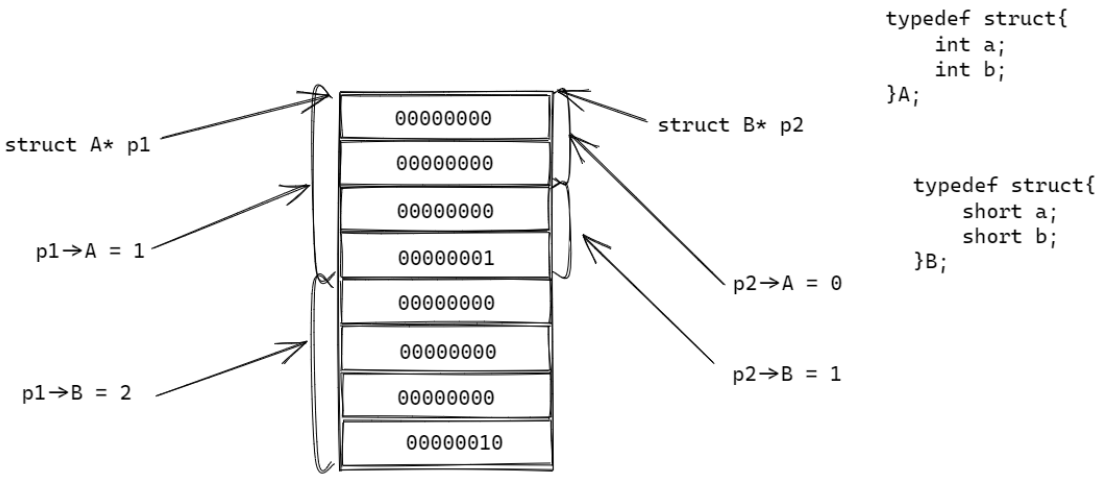
实验代码`/kern/mm/default_pmm.c`中的`default_init()`函数对`free_area_t`结构体`free_area`进行了初始化。

```
free_area_t free_area;

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

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

结构体指针强制转换

与`free_area`的创建方式不同，我们在代码中并没有定义任何一个Page结构体变量。而是通过定义Page结构体指针并将指针指向对应的内存空间地址的形式，通过强制转换将内存中的内容转换为对应的结构体内容。因此内核创建的free_area变量实际存在于kernel所占用的内存区间，而Page结构体是直接指向待分配内存区间的前几页。

下图演示了两个不同的结构体指针指向同一个地址后，结构体中变量的取值：



第三步. 实现页面分配算法

下面介绍一下如何实现一个firstfit内存分配算法的大致流程。firstfit内存分配算法原理很简单，但要在ucore中实现，需要充分了解和利用ucore已有的数据结构和相关操作、关键的一些全局变量等。

现有代码中已经实现了`first_fit`分配算法，该算法需要维护一个查找有序（地址按从小到大排列）空闲块（以页为最小单位的连续地址空间）的数据结构，而双向链表是一个很好的选择。

`default_init_memmap()`根据`page_init`函数中传递过来的参数（某个连续地址的空闲块的起始页，页个数）建立了一个连续内存空闲块的双向链表。这里有一个假定`page_init`函数是按地址从小到大的顺序传来的连续内存空闲块的。链表头是`free_area.free_list`，链表项是Page数据结构的`base->page_link`。这样我们就依靠Page数据结构中的成员变量`page_link`形成了连续内存空闲块列表。

通过空闲块列表，在`default_alloc_pages()`中实现了firstfit的内存分配算法，目前代码尚未涉及进程的运行，因此分配的方式暂时只是将分配页面打上"已用"的标记。具体实现请自行阅读。

第四步. 实现已占用内存页面的释放

与页面分配算法的实现过程相类似，`default_free_pages`实现了使用完毕的页面的释放。请自行阅读。

七、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. 可用内存空间的获取方法
2. 简单的内存分配算法及其实现

八、下一实验简单介绍

在下一个实验中，我们将完成内存管理的剩余部分，使用多级页表，处理缺页异常（Page Fault），实现页面置换算法。