操作系统实验 Lab 2

17343050 黄昱珲 软件工程 2 班

实验目的

- 理解基于段页式内存地址的转换机制
- 理解页表的建立和使用方法
- 理解物理内存的管理方法

练习 1 - 实现 first-fit 连续物理内存分配法算法

`struct Page`

First-Fit 算法维护一个 `free_list` 空闲页帧块列表，这个列表中的每个元素都是一个 `struct Page` 结构体。空闲页帧列表的每个元素都存储这个空闲页帧块的第一个页面。为了区分页帧是不是页帧块中的第一个页帧，我们将 `struct Page` 的 `property` 属性用于标记空闲页帧块的页帧数；`flags` 属性中的 `property` 位用来标记这个页帧是不是页帧块的头页面。因为 `free_list` 的节点都是空闲页帧块的第一个页面，因此所有 `free_list` 节点的 `property` 位均必须为 1，且节点的 `property` 属性就用来存储页帧数。也就是说，我们将一个空闲页帧块的信息都存储在头页面中，`free_list` 以存储这些头页帧的形式来维护所有的空闲页帧块。

```
/**
 * 页描述符。每个页描述一个页帧。
 *
 * 在 kern/mm/pmm.h 中包含了很多页管理的工具函数。
 */
struct Page {
    int ref;                // 页帧的引用计数器，若被页表引用的次数为 0，那么这个页帧将被释放
    uint32_t flags;         // array of flags that describe the status of the page frame
    unsigned int property;  // 不同内存管理算法用作不同用处
    list_entry_t page_link; // 空闲块列表 free_list 的链表项
};
```

`default_init`

First-Fit 算法只需要将 `free_list` 初始化即可。`free_list` 的结构是一个双向的循环链表，`free_area` 为该列表的虚拟头节点。因此 `list_init` 函数将会把 `free_list` 初始化为自己指向自己的单节点双向循环链表。同时需要初始化空闲页帧数 `free_area.nr_free` 为零。

```
/**
 * 维护记录空闲页的双向链表
 *
 * 在初始情况下，也许这个物理内存的空闲页帧都是连续的，这样就形成了一个大的连续
 * 内存空闲块。但随着页帧的分配与释放，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页帧是连续的。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，便于分配和释放，为此定义了一个 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页帧。
 */
typedef struct {
    list_entry_t free_list;         // 空闲块双向链表的虚拟头节点
    unsigned int nr_free;           // 空闲块的总数（以页为单位）
} free_area_t;

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

`default_init_memmap`

这个函数的功能是接收 `pmm.c` 中的 `init_memmap` 函数经过筛选后的连续的可用内存段，并将其加入 `free_list`。由于内存分配器接管了这些内存页面，这些页面就不再是 `reserved` 的了，因此我们首先需要清除这些页面的 `reserved` 标记。我们的 FF 算法实现将 `struct Page::property` 项用于表示一个连续的可用页帧块的头页面，以及这个可用连续页帧块的页面数。这样我们就无须额外申请内存空间来管理这些内存。因此我们在函数开始时也要将每个页面的 `property` 标记清空，防止出错。

初始化页面完成之后，我们就需要将这些可用页面加入 `free_list` 参与内存管理，这些页面由于都未被分配，因此其引用数要初始化为 0。

将这个连续的页帧块加入 `free_list` 时需要将头页面的 `property` 置位，标记头页面并且记录页帧数。而且注意要将页面加入到 `free_list` 的合适位置：由于我们要保证 `free_list` 内容是按地址顺序排列的，`init_memmap` 初始化页帧块时就必须保证这个性质。由于 `BIOS-E820` 中断返回的数据是按照地址排列的，因此我们只需要将数据加入到 `free_list` 末尾即可。

```
/**
 * 将一段连续的物理内存加入到空闲块表中。
 * 将初始化所有页面。
 *
 * @param base 空闲块的头页面
 * @param n 空闲块的页面数，在页表中连续
 *
 * @note 调用链:
 * kern_init -> pmm_init -> page_init -> init_memmap -> pmm_manager ->
pmm_manager.init_memmap
 */
static void default_init_memmap(struct Page *base, size_t n) {
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p)); // 我们在 pmm.c 中将所有页面设置为保留态待分配
        p->flags = p->property = 0; // ClearPageReserved && ClearPageProperty
        set_page_ref(p, 0); // 这些页面都未被使用，因此引用数为 0
    }
    // base 页为该连续空闲页面的头页面
    base->property = n;
```

```
    SetPageProperty(base); // 将 base 标记为头页面
    nr_free += n; // 维护空闲页帧数
    list_add_before(&free_list, &(base->page_link));
}
```

default_alloc_pages

这个函数的功能是分配一段连续的内存空间给用户。我们需要保证 free_list 是顺序的，因此分配内存时只需要按顺序遍历 `free_list` 找到足够大的连续页帧即可。顺序遍历的时间复杂度在最坏情况下是 $O(N)$ 的。

对于这个新找到的页帧，我们需要检查其大小。如果其大小恰好为 n 个页面，那么很好，不需要将这个空闲段进行切分，不会产生碎片，我们直接返回这个页帧块的首页面即可；如果页帧块过大，那么我们需要分裂这个页帧块，而分裂出来的两个新页帧块中，不用来分配内存的那个页帧块原地插入回链表即可，要分配的页帧块恰好包含 n 个页面，需要从从链表中删除并返回首页面。分裂页帧块的时间复杂度是 $O(1)$ 的。

我们不需要在这里管理页面的引用数，页面的引用数由 `pmm.c` 的 `get_pte` 函数管理。因此我们只需要关心页面分配。

为了保证程序最大限度下的可用性，在无法分配内存时（找不到足够大的连续空闲页帧），直接返回 `NULL` 而不是直接出错。

最后整个算法的时间复杂度在最坏情况下是 $O(N)$。

```
/**
 * 在空闲页面表中寻找第一个大小足够大(>=n)的块，
 * 修改块大小成剩余部分的大小。
 * @return 分配的内存的首地址，若分配失败返回 NULL
 */
static struct Page *default_alloc_pages(size_t n) {
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL; // 选中的块
    list_entry_t *le = &free_list;
    // 遍历所有的空闲块，找到第一个至少有 n 个页面的块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) { // 如果成功选中某个块
        if (page->property > n) {
            // 分裂块
            struct Page *p = page + n;
            p->property = page->property - n;
            // 在原处插入新节点以保证 free_list 是按照页地址顺序存储的
            list_add(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
        nr_free -= n;
        ClearPageProperty(page);
```

```
    }
    return page;
}
```

default_free_pages

这个函数的功能是回收被释放的页面。被回收的页面需要加入 `free_list` 以便下一次分配。由于我们要保证 `free_list` 的元素按地址排放，我们需要先找到插入的位置再予以插入。找到要插入的位置的时间复杂度是 $O(N)$ 的，方法是顺序遍历 `free_list`，找到第一个节点的页面地址大于我们当前要释放的页面的地址，将要插入的节点插入到这个节点的前面即可。如果找不到这样的一个节点，说明当前要插入的节点的页面地址在 `free_list` 中最大，我们需要将这个节点插入到链表的尾部。

需要注意的是，`list_add(&free_list, _)` 函数将 `_` 节点插入到 `free_list` 链表的头部，而 `list_add_before(&free_list, _)` 将节点插入到 `free_list` 链表的尾部。这是因为这个链表是双向链表，虚拟头结点的上一个节点就是链表中的最后一个节点。

如果仅仅这样就完成了实现，那么内存块必然会碎片化而且越来越小，因为我们没有将这些连续但不属于同一个页帧块的页面合并在同一个页帧块中，这样分配时就找不到这个实际上是连续的大页帧块从而导致内存的分配失败。除此之外，如果我们通过 `default_alloc_pages` 内实现查找连续的页帧块也会降低效率：`free_list` 必然越来越长，导致搜索页面的速度越来越慢。因此我们需要及时合并可以合并的块。可以合并的页帧块在 `free_list` 中有 3 种情况：

1. `free_list` 中存在于当前被释放的页帧块左相邻的空闲页帧块
2. `free_list` 中存在于当前被释放的页帧块右相邻的空闲页帧快
3. 情况 1 和 2 同时发生

也就是说，我们在找到要插入的位置后，需要合并的页帧块一定在这个插入后的位置的前驱、后继节点。因此我们只需要在插入完成后判断一下前驱后继节点是否和当前节点是连续的，如果是连续的，我们将这些连续的页面块合并：只需要将地址较小的那个页帧块的长度加大并删除地址较大的那个节点（删除后需要清除相应的头页面的 `property` 属性）即可。

并不存在需要合并的页帧块超过 2 个的情况，因为我们在 `default_alloc_pages` 函数中只会删除某个块，或者将某个块的大小减小，也就是说不会影响 `free_list` 的性质：任两个元素都是不可合并的。而我们在 `default_free_pages` 函数中由于一定会保持这个性质，因此 `free_list` 在插入之前的任两个元素是不可合并的，因此我们在一个离散的数轴上插入一个不会相交的线段后，最多只会合并相邻的两个元素。

合并块的时间复杂度是 $O(1)$。最后这个算法的时间复杂度是 $O(N)$ 的。

在释放页面时，我们需要检查我们将要释放的页面是否*可以*释放，如果调用者的逻辑有误，可能会导致过多地释放：释放了未被分配的内存、未被当前内存管理器托管的内存，或者释放了无权释放的内存。如果出现了这种情况，则应该终止页帧释放，如果有必要我们可以设置错误码方便用户查看问题。但此处，内存管理器调用方需要确保调用的参数是正确的：判断是否有权释放内存的工作应该在系统调用时检查、释放未被分配、托管的内存则是调用者的逻辑有误，而调用者仍然是内核代码，如果内核代码有误，则不应该继续执行下去：操作系统有 bug。无论如何，我们应该直接使内核崩溃并检查代码问题。

```
/**
 * 释放 base 开始的连续 n 个页面。
 * 释放时需要检查 free_list 是否存在相邻的块，如果存在则需要合并。
 */
static void default_free_pages(struct Page *base, size_t n) {
    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);
    list_entry_t *le = list_next(&free_list);
    // 遍历列表，查找是否存在可以合并的块
    // 可以合并的块至多两个，一个在 base 前面，一个在 base 后面
    list_entry_t *entry = &free_list;
    for (; le != &free_list; le = list_next(le)) {
        p = le2page(le, page_link);
        if (base + base->property == p) { // 向后合并
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) { // 向前合并
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
        // 寻找插入的位置，我们需要保证 free_list 内元素是按照地址顺序排列的
        else if (base + base->property < p) {
            entry = &p->page_link;
            break;
        }
    }
    nr_free += n;
    list_add_before(entry, &(base->page_link));
}
```

First-Fit 算法改进

First-Fit 算法要求维护一个集合，其中存储空闲的连续页帧块，且顺序按照地址的顺序存储，每次需要查询是否存在页面数大于 n 的块，并且选择其中地址最小的那个块分配内存。也即是 First-Fit 算法倾向于分配尽量低地址的连续内存块，以便使高地址的内存尽量连续。

之前的实现方式是使用一个双向链表，存储空闲的连续页帧块，按照地址顺序存储，每次查询时从链表头结点开始遍历整个链表，一旦遇到块大小足够分配的块，即刻分配，必要时将块分裂后插入到链表中的原位置；回收页面时，也需要遍历整个链表，找到和其相邻的空闲块进行合并。该算法的时间和空间复杂度最坏情况下都是 $O(n)$

为了优化该算法，我们可以采用**伸展树**数据结构来维护连续空闲页帧块。该算法的空间复杂度最坏情况下也是 $O(n)$ 的，均摊的空间复杂度和原双向链表实现是一致的，且实际的内存使用只会比双向链表实现稍大一点。该算法的时间复杂度是均摊 $O(\log n)$ 的。也就是我们可以在牺牲一些空间复杂度的常数的前提下获得很好的时间复杂度。

原理很简单，伸展树的每个节点仍然是连续页帧块，和双向链表一致。节点除了要记录其两个儿子节点的地址、父亲节点的地址、还有子树中最大连续页帧块的页面数；节点的左子树中的所有节点的地址均比该节点的地址小；右子树中的所有节点的地址均比该节点的地址大（也就是伸展树的中序遍历结果和双向链表实现的顺序遍历结果一致）。

查询时我们需要找连续页面数至少为 n 的节点，那么我们只需要递归查询 Splay：每遇到一个节点，该节点的子树中的最大连续页帧块的页面数大于 n，那么我们要找的节点一定在该节点中，我们再判断两个儿子节点：如果左儿子的最大值大于 n，那么就递归左儿子，否则递归右儿子（若左儿子没有，则一定在右儿子中）。优先查找左儿子的目的是查找满足连续页面数大于 n 的所有节点中地址最小的那个。这样因为伸展树的深度是均摊 $O(\log n)$ 的，所以查询算法的复杂度是均摊 $O(\log n)$。

释放页面时，我们可以用 $O(\log n)$ 的时间找到可以合并的某个相邻页面，由于我们可以在 $O(1)$ 内找到所有可以合并的节点（通过 `pages` 数组，且可以合并的节点至多两个），因此，合并后（或者找不到相邻页面，创建新节点）再将节点插入回伸展树即可。

伸展树本身实现不是很复杂，200 行代码内可以完成所有功能。

练习2：实现寻找虚拟地址对应的页表项

`get_pte`

这个函数的作用是通过一级页表（页目录表）以及线性地址获取地址所属的二级页表项。首先我们查找一级页表项（线性地址的高 10 位为一级页表索引），如果这个页目录表项未初始化，我们则需要初始化：分配并初始化（这里直接清零）一个页面给对应的二级页表，同时设置这个页目录表项指向这个二级页表所在的页帧，并设置一级页表项的权限是用户可读写的。经过设计一级页表和二级页表均能恰好存进一个页面中：一个页表项是 32 位，4 个字节，一个页表恰好有 2^{10} 个项（因为无论是线性地址中的页目录索引还是页表索引都是 10 位），因此一个页表的大小恰好为 4KB。最后我们再根据线性地址获取二级页表项：

1. 一级页表项通过地址高 10 位索引 `PDX(la)`；
2. 通过索引到的一级页表项 `*pdep = pgdir[PDX(la)]` 获得二级页表地址的物理地址 `pa = PDE_ADDR(*pdep)`；
3. 由于我们是虚拟地址寻址，因此需要拿到二级页表的虚拟地址 `pt = KADDR(pa)`；
4. 二级页表项通过地址中间 10 位索引 `PTX(la);`
5. 拿到二级页表项地址 `pt[PTX(la)]`。

对二级页表清零是很有必要的：我们通过 `pte & PTE_P` 来检查页面是否存在，如果二级页表所在的页面是脏的，`PTE_P` 位就有可能为 1，从而导致错误，误以为存在二级页表项。

需要注意的是，我们在将二级页表清零时，首先我们可以通过二级页表所在页面号拿到页面的物理地址，而内核是运行在虚拟地址模式下的，因此我们在清零时，需要将页面先调用 `KADDR` 宏转换为虚拟地址再进行清零。

```
/**
 * 根据线性地址以及页目录表获取页表项
 * @param pgdir 内核页表的地址（内核虚地址）
 * @param la 需要查找所在页的线性地址
 * @param create 是否需要分配新的页给二级页表
 * @return 页表项的内核虚拟地址
 */
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    // LAB2 EXERCISE 2: YOUR CODE
    pde_t *pdep = &pgdir[PDX(la)];              // (1) find page directory entry
    if (!(*pdep & PTE_P)) {                     // (2) check if entry is not present, 从
page_insert 中复制而来
        if (create) {                           // (3) check if creating is needed
            // CAUTION: this page is used for page table, not for common data page
            struct Page *page = alloc_page();   // then alloc page for page table
```

```
            if (!page) return NULL;             // 无法分配一个新页用于存储页表项
            set_page_ref(page, 1);              // (4) set page reference
            uintptr_t pa = page2pa(page);       // (5) get physical address of page
            memset(KADDR(pa), 0, PGSIZE);       // (6) clear page content using memset
            *pdep = pa | PTE_USER;          // (7) set page directory entry's permission
        } else {
            return NULL;                        // 无法在没有且不创建页的情况下分配内存
        }
    }
    pte_t *pt = (pte_t *)KADDR(PDE_ADDR(*pdep));
    return &pt[PTX(la)];                        // (8) return page table entry
}
```

请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对 ucore 而言的潜在用处

`mmu.h` 中详细描述了线性地址各部分的含义：

```
+-------10-------+-------10-------+---------12----------+
| Page Directory |   Page Table   | Offset within Page  |
|      Index     |     Index      |                     |
+----------------+----------------+---------------------+
 \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
  \---------- PPN(la) ----------/
```

`memlayout.h` 中描述了 PDE 和 PTE 的类型为 `uintptr_t`，是个 32 位整数，通过 `pmm.c` 中的处理可知：

```
// 页表项类型，高 20 位为页面编号（Address of 4KB page frame; Address of page table）
//            低 12 位为页表项标记，见下图以及 mmu.h 中的 PTE 系列宏。
// 由于二级页表项只能指向某个页面的首地址（物理地址），这个首地址的低 12 位必全零，因此我们可以利用
// 低 12 位来存储一些标记。如果需要获得页面地址，只需要 PTE_ADDR 宏即可（就是把低 12
// 位清零，就可以得到页面地址了）
typedef uintptr_t pte_t;

// 页表目录项类型，高 20 位为二级页表的页面编号，低 12 位为页表项标记，见 mmu.h 中的
// PTE 系列宏。
// 由于页表目录项也是特殊的页表项，因此 pde_t 的结构和 pte_t 一致。
typedef uintptr_t pde_t;
```

PDE 和 PTE 的结构在 *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3A* 中有详细说明（因为页表项需要由 CPU 直接访问，因此 ucore 的页表项结构必须符合 Intel 官方手册）：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored				PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

与上图一致的页表标记在 `mmu.h` 中也描述的很清楚了：

```
#define PTE_P     0x001 // 页面是否存在（是否分配）
#define PTE_W     0x002 // 页面是否可写 R/W
#define PTE_U     0x004 // 页面是否可以在用户态（CPL=3）操作 U/S
#define PTE_PWT   0x008 // 页面是否写穿透
#define PTE_PCD   0x010 // 页面是否禁止载入高速缓存
#define PTE_A     0x020 // 页面是否被访问过
#define PTE_D     0x040 // 页面是否被写入过
#define PTE_PS    0x080 // 页面大小（0 表示二级页表是 4KB 的页）
#define PTE_MBZ   0x180 // PAT&G
#define PTE_AVAIL 0xE00 // 9~11 位提供给操作系统或用户程序进行自由发挥
```

下表是 PDE 各位的功能：

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

下表是 PTE 的各位功能：

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

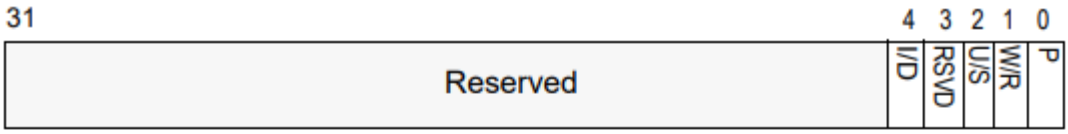
其中，这些标记用于 Lab 3 的开发（实现页交换）：

1. `PTE_PWT` 标记页面是否写穿透，如果页面要求写穿透，那么磁盘上的相应页面交换进内存并发生了写入后，必须同时写入硬盘，这样可以提升页面数据的可靠性，在断电后数据可以恢复。
2. `PTE_D` 项标记页面是否被写入过，如果被写入过，交换进磁盘时就需要一次回写。

如果 ucore 执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情

根据 *Intel® 64 and IA-32 Architectures Software Developer 's Manual 2007 – Volume 3A*，在发生页访问异常时，处理器会启动缺页异常处理器：

1. 向栈内压入如下图的错误码 (Page-Fault error code)。错误码包括如下信息：



P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	The access causing the fault originated when the processor was executing in supervisor mode.
	1	The access causing the fault originated when the processor was executing in user mode.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by reserved bits set to 1 in a page directory.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.

2. 处理器将访问异常的线性地址存入 CR2 寄存器中，以便查找对应的页目录项和页表项。如果页访问异常因权限不足引发，处理器将修改 PDE 的 PTE_A 位（这个行为只会在 Intel 处理器中完成）。
3. 调用 14 号中断（Page-Fault Exception）。

练习3：释放某虚地址所在的页并取消对应二级页表项的映射

`page_remove_pte`

这个函数负责将线性地址所属的页面从页表中删除。之前说过 `pmm.c` 中需要维护页面的引用数，因此如果我们要移除二级页表项，就需要将页帧被页表引用的次数减一。如果引用次数减为 0，这个页帧就不可能再被任何程序使用，应该调用 `free_page` 函数予以回收。且需要通知硬件 TLB 更新内容。

```
// 从页表中删除线性地址 la 所属的页。并更新 TLB。
static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    // LAB2 EXERCISE 3: YOUR CODE
    if (*ptep & PTE_P) { // (1) check if this page table entry is present
        struct Page *page = pte2page(*ptep); // (2) find corresponding page to pte
        if (!page_ref_dec(page)) // (3) decrease page reference
            free_page(page); // (4) and free this page when page reference reachs 0
        *ptep = 0; // (5) clear second page table entry
        tlb_invalidate(pgdir, la); // (6) flush tlb
    }
}
```

数据结构 Page 的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是？

有对应关系。首先 `pages` 的每一项（页面）的下标是这个页面的页号。也就是说，页表的页目录项和页表项指向的页面编号是 `pages` 数组元素的下标。同时，我们在获得一个页表项后，已知页面编号，我们就可以拿到页面对应的 `struct Page` 的实例（`pa2page(pde/pdt) => pages[PPN(pde/pdt)]]`）。

如果希望虚拟地址与物理地址相等，则需要如何修改 lab2，完成此事?

`kernel.ld`

首先需要将 `kernel.ld` 中的

```
SECTIONS {
    . = 0xC0100000;
    /* ... */
}
```

改为

```
SECTIONS {
    . = 0x00100000;
    /* ... */
}
```

因为 `0xC0100000` 是虚拟地址；`0x00100000` 是对应的物理地址。

`memlayout.h`

需要将

```
#define KERNBASE            0xC0000000
```

改为

```
#define KERNBASE            0x00000000
```

因为 `KERNBASE` 表示映射后的物理内存地址，因此将其改为 0 就可以自动将使用虚拟内存地址的代码改用物理地址。

实验心得

通过这次实验，我对 x86 体系结构下的物理内存管理机制（分页分段）、以及页分配算法有了基本的了解：

1. bootloader 会调用 BIOS e820 中断来获取系统可用的物理内存区间，为内核进行页帧划分提供基础。
2. 操作系统进入虚拟内存寻址的方式：开启保护模式，创建启动段表→进入分页模式，加载初始页目录表→跳转到高地址→更新初始页目录表→完善段表和页表→进入虚拟寻址模式；这个进入虚拟寻址模式的步骤令我印象深刻，因为这样做充分保证了执行过程的稳定性和正确性。
3. First-Fit 算法的实现方式，如何分配及回收页帧，如何减少外碎片的产生。缺点是容易造成低地址的碎片、查找可用页面的速度比较慢、容易出现页面分配失败的情况；优点是代码易于编写，实现简单。
4. 伙伴系统算法的实现方式，如何分配和回收页帧，如何减少内外碎片的产生。优点是页面分配的速度快：最坏情况下也只要 $O(\log n)$ 的时间、外碎片少；缺点是内碎片可能会很多，如果要解决问题需要用户程序尽量申请

接近或刚好 2 的幂次的内存区块。

5. 操作系统如何根据页表及线性地址来查找物理地址。
6. 如何实现页面共用：通过维护页面的 ref 来实现共享页面的回收。
7. 了解了 `kernel.ld` 的各部分代码的意义和目的，知道了链接器脚本的基本语法。
8. 了解了如何在物理地址、虚拟地址、线性地址并存的情况下正确处理指针。

这次实验让我在实践中加深了对 First-Fit 算法的实现机制的印象，同时通过思考如何优化 First-Fit 算法加深了对该算法的了解：其优缺点是什么。