report

练习题1

- split_chunk先判断chunk的order是否符合要求，如果符合则返回，如果不符合则进行递归调用选择order-1的新chunk，在这个过程中因为对原chunk分裂会产生新的chunk，所以要更新对应的free_list
- merge_chunk先判断当前对象的buddy。如果不满足相应条件就在order+1后继续递归，直到阶数达到上限
- buddy_get_pages中先确认当前order是否有free_list的元素，若没有则在order+1中继续寻找。找到后取出第一个page，在对其split_chunk以获取需要的
- buddy_free_pages，将其与buddy合并后放回到free_list中去

练习题2

- choose_new_current_slab，将pool的current指向patial_slab_list的第一个slab，然后从patial_slab_list中去除该元素
- alloc_in_slab_impl,获取current_slab中的free_list_head,如果可用的slot数量为零就调用choose_new_current_slab寻找新的slab以获取新的slot
- free_in_slab,将当前slot_list指针添加到current_slab的free_list_head头部，然后连接上原来free_list_head的next，最后更新当前free的slot个数。

练习题3

- 根据分配的大小分别调用buddy或者slab中对应的函数

练习题4

- query_in_pgtbl中，遍历三级页表后，在第三级页表中返回物理地址以及入口
- map_range_in_pgtbl_common，遍历三级页表，在最后一级页表后设置好pte的相关属性，将其添加到页表中，完成映射
- unmap_range_in_pgtbl_common，遍历三级页表，在最后一级页表后将pte的有效位置零
- mprotect_in_pgtbl进行遍历翻译后，修改pte的权限

思考题5

在操作系统中支持写时拷贝（Copy-on-Write，CoW）需要配置页表描述符。三级页表项的 AP 字段规定了该物理页的读写权限。为支持写时拷贝，需要将 AP 字段置为 11 ，设为只读权限，当对该区域进行写入时，就会触发异常，此时，操作系统会将异常的物理页拷贝一份，并把 AP 字段置为 01 ，此时权限可读可写，再回到原进程继续操作。

思考题6

直接使用启动时的粗粒度页表，可能会导致较多内存的浪费

挑战题7

我们注意到原程序中mmu.c中采用的是2M的粗粒度，截取其中一段代码如下

```
        /* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
        /* Peripheral memory: PERIPHERAL_BASE ~ PHYSMEM_END */
        /* Map with 2M granularity */
        for (vaddr = KERNEL_VADDR + PERIPHERAL_BASE;
             vaddr < KERNEL_VADDR + PHYSMEM_END;
             vaddr += SIZE_2M) {
                boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                        (vaddr - KERNEL_VADDR) /* high mem, va = kbase + pa */
                        | UXN /* Unprivileged execute never */
                        | ACCESSED /* Set access flag */
                        | NG /* Mark as not global */
                        | DEVICE_MEMORY /* Device memory */
                        | IS_VALID;
        }
```

此时我们需要调用map_range_in_pgtbl(*pgtbl, va, pa, len, flags) 函数，分别以4KB粒度映射 PHYSMEM_START ~ PERIPHERAL_BASE 和 PERIPHERAL_BASE ~ PHYSMEM_END 范围的物理地址到内核页表中。

```
        map_range_in_pgtbl_kernel(
            (void*)((unsigned long)boot_ttbr1_l0),
            KERNEL_VADDR + PERIPHERAL_BASE,
            PERIPHERAL_BASE,
            PHYSMEM_END-PERIPHERAL_BASE,
                        UXN
                        | ACCESSED
                        | NG
                        | INNER_SHARABLE
                        | NORMAL_MEMORY
                        | IS_VALID,
                        NULL);
```

练习题8

调用handle_trans_fault传递缺页异常。

练习题9

调用rb_search和rb_entry进行搜索，获取vaddr的vmr.

练习题10

调用函数对pmo_shm和pmo_anonym情况完成物理页分配及映射。

挑战题11

尝试实现FIFO策略，以下是设置的数据结构

```
typedef struct PageNode {
    paddr_t page_addr;
    struct PageNode *next;
} PageNode;


typedef struct {
    PageNode *head;
    PageNode *tail;
    int size;
} PageQueue;


PageQueue globalPageQueue = {NULL, NULL, 0};

void initPageQueue() {
    globalPageQueue.head = NULL;
    globalPageQueue.tail = NULL;
    globalPageQueue.size = 0;
}

void enqueuePage(paddr_t page_addr) {
    PageNode *newNode = malloc(sizeof(PageNode));
    newNode->page_addr = page_addr;
    newNode->next = NULL;

    if (globalPageQueue.tail) {
        globalPageQueue.tail->next = newNode;
    }
    globalPageQueue.tail = newNode;

    if (globalPageQueue.head == NULL) {
        globalPageQueue.head = newNode;
    }

    globalPageQueue.size++;
}

paddr_t dequeuePage() {
    if (globalPageQueue.head == NULL) {
        return 0;
    }

    PageNode *temp = globalPageQueue.head;
    paddr_t page_addr = temp->page_addr;

    globalPageQueue.head = globalPageQueue.head->next;
    if (globalPageQueue.head == NULL) {
        globalPageQueue.tail = NULL;
    }
```

```
    free(temp);
    globalPageQueue.size--;

    return page_addr;
}
```