练习题1

`split_chunk`函数中，我们采用了非递归的方式（大意了没看到注释），每次从空闲链表中删除原有的chunk，再加入两个低一阶的chunk。

`merge_chunk`则反过来，只要`buddy_chunk`为`NULL`，`allocated`或`order`不相同，就直接在空闲链表中加入一个entry然后返回；否则就可以继续合并，这里使用了递归。

`buddy_get_pages`中只需要找到正确的`order`然后调用`split_chunk`函数即可。

`buddy_free_pages`中只需要将`page->allocated`设置为0再调用`merge_chunk`函数即可。

练习题2

在`slab.c`中，我们需要管理的接口仅有全局变量`struct slab_pointer slab_pool`，处理与之相关的`current_slab`和`partial_slab_list`。因此我们需要考虑的内容仅有两方面：

- 管理`current_slab`中的空闲链表、空闲slot个数`current_free_cnt`；
- 管理`partial_slab_list`。

只需要按照注释的要求实现三个函数的功能即可。

（对于`alloc_in_slab_impl`函数，我们及时调用`choose_new_current_slab`函数确保`current_slab`在任何时刻都是有空闲的，而不是在需要新slab时再分配。）

练习题3

有buddy system和slab的实现之后，只需要对不同的大小分别调用`alloc_in_slab`函数和`get_pages`函数即可。

练习题4

在页表管理中，我们会反复用到以下相同的代码段：

```
        s64 total_page_cnt;
        ptp_t* l0_ptp = (ptp_t*) pgtbl, *l1_ptp = NULL, *l2_ptp = NULL, *l3_ptp = NULL;
        pte_t* pte;
        int ret, pte_index, i;

        BUG_ON(pgtbl == NULL);
        BUG_ON(va % PAGE_SIZE);
        total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1 : 0);
```

```
        while (total_page_cnt > 0) {
                ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, true, NULL);
                BUG_ON(ret != 0);
                ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, true, NULL);
                BUG_ON(ret != 0);
                ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, true, NULL);
                BUG_ON(ret != 0);

                pte_index = GET_L3_INDEX(va);
                ... // functions
        }
```

但是由于`query_in_pgtbl`函数要求我们在出现错误时返回`-ENOMAPPING`，并且其功能意味着需要判断`get_next_ptp`函数的结果是`ptp_t`还是`pte_t`，我们需要额外的实现代码如下：

```
        ptp_t* current_ptp = (ptp_t*) pgtbl; pte_t* pte;
        int query_result;
        for (int i=0; i<4; i++) {
                query_result = get_next_ptp(current_ptp, i, va, &current_ptp,
                                            &pte, false, NULL);
                if (query_result == -ENOMAPPING) return query_result;
                else if ((query_result == BLOCK_PTP || i==3) && i != 0) {
                        if (entry != NULL) *entry = pte;
                        switch (i) {
                                case 1:
                                *pa = virt_to_phys(current_ptp) + GET_VA_OFFSET_L1(va);
                                return 0; break;
                                case 2:
                                *pa = virt_to_phys(current_ptp) + GET_VA_OFFSET_L2(va);
                                return 0; break;
                                case 3:
                                *pa = virt_to_phys(current_ptp) + GET_VA_OFFSET_L3(va);
                                return 0;
                        }
                }
        }
```

（这里我以为循环能够节省代码量、节省时间，但是debug导致了大量的时间浪费）

于是对于剩下的函数，我们只需要实现给定`pte_index`之后的功能即可。对于`map_range_in_pgtbl_common`函数，我们需要重复创建页表映射，管理`pte_t`数据结构的各个属性；对于`unmap_range_in_pgtbl`函数，我们只要将`is_valid`设置为false即可；`mprotect_in_pgtbl`同理，只需要调用`set_pte_flags`函数。

思考题5

观察ChCore代码，支持CoW只需要配置权限`vmr->perm`为`VMR_READ | VMR_COW`即可。

由于CoW实际上是触发页的权限错误，因此处理page fault时从`pagefault.c`中的`do_page_fault`函数跳转到`pagefault_handler.c`中的`handle_perm_fault`函数，在函数中比较`declared_perm`和`desired_perm`的区别，判断是否为CoW。若是CoW，则分配新的页，复制原页内容，并把新的页`perm`属性设置为`pte_info->perm | VMR_WRITE`。

实际实现中，ChCore是通过将`page_table.h`中定义的`AP`字段使用`pare_pte_to_common`函数转化为`...->perm`的形式实现的（应该是为了以architecture independent的形式实现CoW所以这么做的？）。

思考题6

映射2M的大页产生的内部碎片会显著地多于使用4K的小页。

挑战题7

我已经有一个绝妙的实现方法，但是时间太短来不及了。（并没有）

练习题8

如题所述，只需要`ret = handle_trans_fault(current_thread->vmspace, fault_addr);`即可。

练习题9

我们无需关注红黑树的实现，只需要调用`rb_search`搜索`vmr_tree`再用`rb_entry`将`rb_node`形式的返回值改为所属的`vmregion`返回即可。

练习题10

代码段1：物理地址没有记录。只需根据注释提示的内容，使用`get_pages`函数分配一个物理页并使用`memset`清除内容即可。但是注意到后文需要将physical page记录到radix tree中，因此需要知道该页的物理地址，使用`virt_to_phys`函数将结果保存到`pa`中即可。

代码段2：物理地址在PMO中已经记录，则不应该重复分配物理页，调用`map_range_in_pgtbl`即可。