练习题 1:

merge_chunk:

```
struct page *buddy = NULL;
if (chunk->order == BUDDY_MAX_ORDER - 1) {
        return chunk;
}

buddy = get_buddy_chunk(pool, chunk);

if (buddy == NULL) {
        return chunk;
}

if (buddy->allocated == 1) {
        return chunk;
}

if (buddy->order != chunk->order) {
        return chunk;
}

list_del(&(buddy->node));

pool->free_lists[buddy->order].nr_free--;
buddy->order++;
chunk->order++;
if (chunk > buddy) {
        chunk = buddy;
}

return merge_chunk(pool, chunk);
```

split_chunk:

合并参考教程，采取递归方式，先判断能否进行递归调用，如果不能则直接返回该 chunk，删除 list 中的 buddy 并更新相关信息，在小块合并后，如果大块能然能合并就会再次合并

分裂使用循环的方式，如果不是需要的 order，就一直向下分裂，直到获取所需大小的 chunk，期间需要加入对应 order 的 free_list

```
if (chunk == NULL || chunk->allocated == 1)
        return NULL;

struct page *buddy = NULL;
while (order != chunk->order) {
        chunk->order--;
        buddy = get_buddy_chunk(pool, chunk);
        buddy->allocated = 0;
        buddy->order = chunk->order;
        pool->free_lists[chunk->order].nr_free++;
        list_add(&(buddy->node), &(pool->free_lists[chunk->order].free_list));
}

BUG_ON(chunk == NULL);
return chunk;
```

buddy_get_pages

```
for (cur_order = order; cur_order < BUDDY_MAX_ORDER; cur_order++) {
        free_list = &(pool->free_lists[cur_order].free_list);
        if (pool->free_lists[cur_order].nr_free > 0) {
                page = list_entry(free_list->next, struct page, node);
                pool->free_lists[cur_order].nr_free--;
                list_del(&(page->node));
                if (page == NULL)
                        return NULL;
                page = split_chunk(pool, order, page);
                if (page == NULL)
                        return NULL;
                page->allocated = 1;
                //return page;
                break;
        }
}
```

buddy_free_pages

```
page = merge_chunk(pool, page);
page->allocated = 0;
order = page->order;
free_list = &(pool->free_lists[order].free_list);
pool->free_lists[order].nr_free += 1;
list_add(&page->node, free_list);
```

获取页从低到高遍历 order，如果当前 order 存在 free 的部分，就删除结点，进行拆分，没有则继续向后寻找。
释放页则直接进行合并，然后放回 free_list

练习题 2:

choose_new_current_slab

```
struct list_head *list = &(pool->partial_slab_list);
if (list_empty(list)) {
        pool->current_slab = NULL;
} else {
        pool->current_slab = (struct slab_header *)list_entry(
                list->next, struct slab_header, node
        );
        list_del(list->next);
}
```

alloc_in_slab_impl

```
free_list = (struct slab_slot_list *)current_slab->free_list_head;
// BUG_ON
next_slot = free_list->next_free;
current_slab->current_free_cnt--;
current_slab->free_list_head = next_slot;
if (unlikely(current_slab->current_free_cnt == 0))
        choose_new_current_slab(&slab_pool[order]);
```

free_in_slab

```
/* BLANK BEGIN */
slot->next_free = slab->free_list_head;
slab->free_list_head = slot;
slab->current_free_cnt++;
/* BLANK END */
```

1. Choose… 将 pool->current_slab 指向空闲 slab_list 中的第一个 slab，然后删除掉该 slab，表示已经被使用
2. Alloc… 获取当前 slab 中头部的空闲 slot(free_list_head)，更新相关信息，如果当前 slab 空闲 slot 为 0，则获取新的
3. Free… 将该 slot 的 next 指向原头部 free_list_head，slab 的 free_list_head 指向该 slot，最后增加 free 的数量

练习题 3:

```
 if (size <= SLAB_MAX_SIZE) {
        /* LAB 2 TODO 3 BEGIN */
        /* Step 1: Allocate in slab for small requests. */
        /* BLANK BEGIN */
        addr = alloc_in_slab(size, real_size);
        /* BLANK END */
ABLE_MEMORY_USAGE_COLLECTING == ON
        if(is_record && collecting_switch) {
                record_mem_usage(*real_size, addr);
        }

 } else {
        /* Step 2: Allocate in buddy for large requests. */
        /* BLANK BEGIN */
        if (size <= BUDDY_PAGE_SIZE)
                order = 0;
        else
                order = size_to_page_order(size);
        addr = get_pages(order);
        /* BLANK END */
        /* LAB 2 TODO 3 END */
 }
```

如果小于 slabsize，就从 slab 分配内存，否则根据内存大小判断 order，在 buddy 中分配内存

练习题 4:

```
ptp_t *cur_ptp = (ptp_t *)pgtbl;
int check_ret;
for(int idx = 0; idx <= 3; idx++)
{
        ptp_t *next_ptp;
        pte_t *cur_entry;
        check_ret = get_next_ptp(cur_ptp, idx, va, &next_ptp, &cur_entry, false, NULL);
        if (check_ret < 0)
                return -ENOMAPPING;
        if (check_ret == BLOCK_PTP && idx != 3) {
                switch (idx) {
                        case 1:
                                *pa = virt_to_phys((vaddr_t) next_ptp) + GET_VA_OFFSET_L1(va);
                                break;
                        case 2:
                                *pa = virt_to_phys((vaddr_t) next_ptp) + GET_VA_OFFSET_L2(va);
                                break;
                        default:
                                break;
                }
                if(entry != NULL)
                        *entry = cur_entry;
                return;
        }
        if(idx == 3)
        {
                *pa = GET_PADDR_IN_PTE(cur_entry)+ GET_VA_OFFSET_L3(va);
                if(entry != NULL)
                        *entry = cur_entry;
                break;
        }
        cur_ptp = next_ptp;
}
```

遍历三级页表，在第三级页表处返回对应的物理地址以及 pte entry，如果 1，2 级页表是大页，也返回对应的数据

```
s64 total_page_cnt;
ptp_t *l0, *l1, *l2, *l3;
pte_t *pte;
int ret, idx, i;
total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1 : 0);
l0 = (ptp_t *)pgtbl;
l1 = NULL;
l2 = NULL;
l3 = NULL;

while(total_page_cnt > 0) {
        ret = get_next_ptp(l0, L0, va, &l1, &pte, true, rss);
        BUG_ON(ret != 0);

        ret = get_next_ptp(l1, L1, va, &l2, &pte, true, rss);
        BUG_ON(ret != 0);

        ret = get_next_ptp(l2, L2, va, &l3, &pte, true, rss);
        BUG_ON(ret != 0);


        // ret = get_next_ptp(l3, L3, va, &l3)
        idx = GET_L3_INDEX(va);
        for (i = idx; i < PTP_ENTRIES; i++) {
                pte_t n_pte;
                n_pte.pte = 0;
                n_pte.l3_page.is_page = 1;
                n_pte.l3_page.is_valid = 1;
                n_pte.l3_page.pfn = pa >> PAGE_SHIFT;
                set_pte_flags(&n_pte, flags, kind);
                l3->ent[i].pte = n_pte.pte;

                va += PAGE_SIZE;
                pa += PAGE_SIZE;
                if (rss) {
                        *rss += PAGE_SIZE;
                }
                total_page_cnt -= 1;
                if (total_page_cnt == 0) {
                        break;
                }
        }
}
```

Map_range_in_pgtbl_common 遍历页表，在最后一级设置相关属性，添加至页表项，完成 4k 粒度的映射

```
check_ret = get_next_ptp(l1, L1, va, &l2, &pte, false, rss);
if (rss && l2 != l2_snapshot) {
        l2_snapshot = l2;
        snapshot -= PAGE_SIZE;
}
if (check_ret < 0) {
        page_num -= L1_PER_ENTRY_PAGES;
        va += L1_PER_ENTRY_PAGES * PAGE_SIZE;
        if (rss) {
                *rss -= L1_PER_ENTRY_PAGES * PAGE_SIZE;
        }
        continue;
}

check_ret = get_next_ptp(l2, L2, va, &l3, &pte, false, rss);
if (rss && l3 != l3_snapshot) {
        l3_snapshot = l3;
        snapshot -= PAGE_SIZE;
}
if (check_ret < 0) {
        page_num -= L2_PER_ENTRY_PAGES;
        va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
        if (rss) {
                *rss -= L2_PER_ENTRY_PAGES * PAGE_SIZE;
        }
        continue;
}

for (int i = GET_L3_INDEX(va); i < PTP_ENTRIES; i++) {
        l3->ent[i].pte = PTE_DESCRIPTOR_INVALID;
        //printk("unmap is %d\n", i);
        page_num--;
        //printk("page num is now %d\n", page_num);
        if (rss) {
                *rss -= PAGE_SIZE;
                snapshot -= PAGE_SIZE;
        }
        if (page_num == 0) {
                break;

        }
        va += PAGE_SIZE;
}
recycle_pgtable_entry(l0, l1, l2, l3, old, rss);
```

<-部分代码片段

Unmap， 遍历页表，在第三级页表的对应位置将 pte 归零

```
u64 page_num = len / PAGE_SIZE + (len / PAGE_SIZE) > 0;
ptp_t *l0 = (ptp_t *)pgtbl, *l1 = NULL, *l2 = NULL, *l3 = NULL;
pte_t *pte;
int check_ret;

while (page_num > 0) {
        check_ret = get_next_ptp(l0, L0, va, &l1, &pte, false, NULL);
        if (check_ret < 0) {
                return -ENOMAPPING;
        }

        check_ret = get_next_ptp(l1, L1, va, &l2, &pte, false, NULL);
        if (check_ret < 0) {
                return -ENOMAPPING;
        }

        check_ret = get_next_ptp(l2, L2, va, &l3, &pte, false, NULL);
        if (check_ret < 0) {
                return -ENOMAPPING;
        }

        for (int i = GET_L3_INDEX(va); i < PTP_ENTRIES; i++) {
                set_pte_flags(&(l3->ent[i].pte), flags, USER_PTE);
                va += PAGE_SIZE;
                page_num--;
                if (page_num == 0) {
                        break;
                }
        }
}
```

Mprotect：遍历页表取到第三级 pte 之后，对权限进行修改

思考题 5:

需要设置访问权限位：AP 字段，将权限设置为只读，当出现写操作时，发生 page fault

如何处理：CPU 将控制流传递给缺页异常处理函数，OS 发现对只读内存进行了写操作，对应区域为 COW，OS 会将发生异常的物理页进行拷贝，把页权限设置为读写，然后更新页表项，此时进程可以正常访问该页进行读写

思考题 6:

内存浪费：进程可能不会完整地使用大页，而未被使用的大页内存也不能分配给其它进程，导致了内存的浪费

权限控制：对大页 pte 设置完权限之后，该块内存的权限都是相同的，无法进行细粒度的权限管理

内存碎片：当大页分配过多时，可能导致页之间产生空隙虽然总量足够，但是单一空隙的大小并不足以进行分配，造成内存碎片

挑战题 7:

```
ALIGN(PAGE_SIZE)
char empty_page[4096] = {0};
```

```
memset(empty_page, 0, PAGE_SIZE);
vmr_prop_t flag1, flag2, flag3;
flag1 = VMR_EXEC;
map_range_in_pgtbl_kernel(empty_page, KBASE, (paddr_t)0, 0x3f000000ul, flag1);

flag2 = VMR_EXEC | VMR_DEVICE;
map_range_in_pgtbl_kernel(empty_page, KBASE + 0x3f000000, (paddr_t)0x3f000000, 0x1000000ul, flag2);

flag3 = VMR_EXEC | VMR_DEVICE;
map_range_in_pgtbl_kernel(empty_page, KBASE + 0x40000000, (paddr_t)0x40000000, 0x40000000ul, flag3);
u64 phy_addr = virt_to_phys(empty_page);
asm volatile("msr ttbr1_el1, %0" : :"r"(phy_addr));

flush_tlb_all();
kinfo("[Chcore] map for 4k finished\n");
```

观察到 main.c 中全局设置了一个空白页，使用该页作为 L0 对 kernel 内存进行映射，映射完毕后通过 msr 指令切换 L0 页表

```
printk("empty page is %lx, ttbr is %lx\n", empty_page, boot_ttbr1_l0);
/* Mapping KSTACK into kernel page table. */
map_range_in_pgtbl_kernel((void*)((unsigned long)empty_page),
    KSTACKx_ADDR(0),
    (unsigned long)(cpu_stacks[0]) - KBASE,
    CPU_STACK_SIZE, VMR_READ | VMR_WRITE);
```

```
map_range_in_pgtbl_kernel((void*)((unsigned long)empty_page),
    KSTACKx_ADDR(cpuid),
    (unsigned long)(cpu_stacks[cpuid]) - KBASE,
    CPU_STACK_SIZE, VMR_READ | VMR_WRITE);

arch_interrupt_init_per_cpu();
```

在对 kernel stack 进行映射的时候，将 L0 替换为 empty_page(需要注意 boot ttbr1 为低地址，需要加上 KBASE，而 empty page 为高地址)

练习题 8:

```
/* BLANK BEGIN */
ret = handle_trans_fault(current_thread->vmspace, fault_addr);
/* BLANK END */
/* LAB 2 TODO 5 END */
```

使用 handle_trans_fault 对异常进行处理

练习题 9:

```
/* LAB 2 TODO 6 BEGIN */
/* Hint: Find the corresponding vmr for @addr in @vmspace */
/* BLANK BEGIN */
struct rb_node *node = rb_search(&vmspace->vmr_tree, (const void *)addr, cmp_vmr_and_va);
struct vmregion *vmr = rb_entry(node, struct vmregion, tree_node);
return vmr;
/* BLANK END */
/* LAB 2 TODO 6 END */
```

使用 rb_search 函数寻找对应地址的 node，再通过 rb_entry 取出结构体的起始地址

练习题 10:

```
/* Hint: Allocate a physical page and clear it to 0. */
void *new_pg = get_pages(0);
BUG_ON(new_pg == NULL);
pa = virt_to_phys(new_pg);
BUG_ON(pa == 0);
memset((void *)phys_to_virt(pa), 0, PAGE_SIZE);
/* BLANK END */
```

```
/* BLANK BEGIN */
map_range_in_pgtbl(vmspace->pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss);
/* BLANK END */
```

```
/* BLANK BEGIN */
map_range_in_pgtbl(vmspace->pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss);
/* BLANK END */
/* LAB 2 TODO 7 END */
```

先分配对应的页，并将其置零

然后对缺少的页进行映射

挑战题 11： 此处修改过 lab2.c 中的接口，否则基础测试的参数不正确(如下图)

```
ret = map_range_in_pgtbl(
        pgtbl, 0x1001000, 0x1000, PAGE_SIZE, flags);
lab_assert(ret == 0);

ret = query_in_pgtbl(pgtbl, 0x1001000, &pa, &pte);
lab_assert(ret == 0 && pa == 0x1000);
lab_assert(pte && pte->l3_page.is_valid && pte->l3_page.is_page
        && pte->l3_page.SH == INNER_SHAREABLE);
ret = query_in_pgtbl(pgtbl, 0x1001050, &pa, &pte);
lab_assert(ret == 0 && pa == 0x1050);

ret = unmap_range_in_pgtbl(pgtbl, 0x1001000, PAGE_SIZE);
```

```
156         ret = map_range_in_pgtbl(
157+                pgtbl, 0x1001000, 0x1000, PAGE_SIZE, flags, NULL);
158         lab_assert(ret == 0);
159
160         ret = query_in_pgtbl(pgtbl, 0x1001000, &pa, &pte);
161         lab_assert(ret == 0 && pa == 0x1000);
162         lab_assert(pte && pte->l3_page.is_valid && pte->l3_page.is_page
163                 && pte->l3_page.SH == INNER_SHAREABLE);
164         ret = query_in_pgtbl(pgtbl, 0x1001050, &pa, &pte);
165         lab_assert(ret == 0 && pa == 0x1050);
166
167+        ret = unmap_range_in_pgtbl(pgtbl, 0x1001000, PAGE_SIZE, NULL);
```

修改 get_next_ptp map unmap 对应参数，增加 rss 参数

get_next_ptp 中，如果新分配页表，相应 rss += PAGE_SIZE

map 函数中，映射多少页，就对 rss 增加响应的大小

unmap 函数中，根据取消映射的页数，减少对应的 rss，同时调用 recycle_pgtable_entry 进行页表页的回收，并添加判断，当 map 时分配页表但 unmap 时不释放页表时，会减去页表部分的 rss，同理当 map 不分配页表，但 unmap 释放页表时，也会加上页表部分的 rss

(下为部分片段)

```
new_ptp = get_pages(0);
if (new_ptp == NULL)
        return -ENOMEM;
memset((void *)new_ptp, 0, PAGE_SIZE);
if (rss)
        *rss += PAGE_SIZE;

new_ptp_paddr = virt_to_phys((vaddr_t)new_ptp);
```

```
idx = GET_L3_INDEX(va);
for (i = idx; i < PTP_ENTRIES; i++) {
        pte_t n_pte;
        n_pte.pte = 0;
        n_pte.l3_page.is_page = 1;
        n_pte.l3_page.is_valid = 1;
        n_pte.l3_page.pfn = pa >> PAGE_SHIFT;
        set_pte_flags(&n_pte, flags, kind);
        l3->ent[i].pte = n_pte.pte;

        va += PAGE_SIZE;
        pa += PAGE_SIZE;
        if (rss) {
                *rss += PAGE_SIZE;
        }
        total_page_cnt -= 1;
        if (total_page_cnt == 0) {
                break;
        }
}
```

```
check_ret = get_next_ptp(l2, L2, va, &l3, &pte, false, rss);
if (rss && l3 != l3_snapshot) {
        l3_snapshot = l3;
        snapshot -= PAGE_SIZE;
}
if (check_ret < 0) {
        page_num -= L2_PER_ENTRY_PAGES;
        va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
        if (rss) {
                *rss -= L2_PER_ENTRY_PAGES * PAGE_SIZE;
        }
        continue;
}

for (int i = GET_L3_INDEX(va); i < PTP_ENTRIES; i++) {
        l3->ent[i].pte = PTE_DESCRIPTOR_INVALID;
        //printk("unmap is %d\n", i);
        page_num--;
        //printk("page num is now %d\n", page_num);
        if (rss) {
                *rss -= PAGE_SIZE;
                snapshot -= PAGE_SIZE;
        }
        if (page_num == 0) {
                break;

        }
        va += PAGE_SIZE;
}
recycle_pgtable_entry(l0, l1, l2, l3, old, rss);
```

Lab 本地评测结果：

```
Succeeded to build all targets
To use chcore toolchain, please input the commnad
        export CMAKE_TOOLCHAIN_FILE=/home/voider/Desktop/OStest/OS-Course-Lab/build/toolchain.cmake
make[1]: Leaving directory '/home/voider/Desktop/OStest/OS-Course-Lab'
===============
Grading lab 2...(may take 10 seconds)
GRADE: Allocate & free order 0: 5
GRADE: Allocate & free each order: 5
GRADE: Allocate & free all orders: 5
GRADE: Allocate & free all memory: OK: 5
GRADE: kmalloc: 10
GRADE: Map & unmap one page: 10
GRADE: Map & unmap multiple pages: 10
GRADE: Map & unmap huge range: 20
GRADE: Compute physical memory-1: 1
GRADE: Compute physical memory-2: 1
GRADE: Compute physical memory-3: 1
GRADE: Compute physical memory-4: 2
GRADE: Page fault: 30
===============
Score: 105/100
voider@voider-virtual-machine:~/Desktop/OStest/OS-Course-Lab$
```