练习题 1：

在 split_chunk 函数中输入以下代码:

```
/* LAB 2 TODO 1 BEGIN */
/*
 * Hint: Recursively put the buddy of current chunk into
 * a suitable free list.
 */
/* BLANK BEGIN */
if (chunk->order == order){
        return chunk;
}

chunk->order -= 1;
struct page *buddy = get_buddy_chunk(pool, chunk);
buddy->allocated = 0;
buddy->order = chunk->order;
list_add(&buddy->node, &pool->free_lists[buddy->order].free_list);
pool->free_lists[buddy->order].nr_free += 1;

return split_chunk(pool, order, chunk);
/* BLANK END */
/* LAB 2 TODO 1 END */
```

该段代码在 chunk 分裂后，更新了其 allocated 和 order 字段，同时在链表中更新空闲块的信息。

在 merge_chunk 函数中输入以下代码:

```
/* LAB 2 TODO 1 BEGIN */
/*
 * Hint: Recursively merge current chunk with its buddy
 * if possible.
 */
/* BLANK BEGIN */
struct page *buddy_chunk;

if (chunk->order == (BUDDY_MAX_ORDER - 1)){
        return chunk;
}

buddy_chunk = get_buddy_chunk(pool, chunk);

if (buddy_chunk == NULL)
        return chunk;

if (buddy_chunk->allocated == 1)
        return chunk;

if (buddy_chunk->order != chunk->order)
        return chunk;

list_del(&(buddy_chunk->node));
pool->free_lists[buddy_chunk->order].nr_free -= 1;

buddy_chunk->order += 1;
chunk->order += 1;
if (chunk > buddy_chunk)
        chunk = buddy_chunk;

return merge_chunk(pool, chunk);
/* BLANK END */
/* LAB 2 TODO 1 END */
```

该段代码与 split_chunk 类似，通过递归的方式，在 chunk 完成合并后更新其 order 字段，同时更新链表信息。
在 buddy_get_pages 函数中输入以下代码：

```
/* LAB 2 TODO 1 BEGIN */
/*
 * Hint: Find a chunk that satisfies the order requirement
 * in the free lists, then split it if necessary.
 */
/* BLANK BEGIN */
for (cur_order = order; cur_order < BUDDY_MAX_ORDER; cur_order++){
        if (pool->free_lists[cur_order].nr_free > 0){
                free_list = &pool->free_lists[cur_order].free_list;
                page = list_entry(free_list->next, struct page, node);
                break;
        }
}

if (page == NULL){
        goto out;
}

list_del(&page->node);
pool->free_lists[cur_order].nr_free -= 1;
page = split_chunk(pool, order, page);
page->allocated = 1;
/* BLANK END */
/* LAB 2 TODO 1 END */
```

该段代码首先遍历链表，找到第一个合适大小的空闲块并分裂为用来分配的内存块。
在 buddy_free_pages 中输入以下代码：

```
/* LAB 2 TODO 1 BEGIN */
/*
 * Hint: Merge the chunk with its buddy and put it into
 * a suitable free list.
 */
/* BLANK BEGIN */
page->allocated = 0;

page = merge_chunk(pool, page);

order = page->order;
free_list = &(pool->free_lists[order].free_list);
list_add(&page->node, free_list);
pool->free_lists[order].nr_free += 1;
/* BLANK END */
/* LAB 2 TODO 1 END */
```

该段代码与前面的 get_pages 相反，将待释放的内存块合并并更新相应的信息。

练习题 2：

在 choose_new_current_slab 函数中填入以下代码：

```
/* LAB 2 TODO 2 BEGIN */
/* Hint: Choose a partial slab to be a new current slab. */
/* BLANK BEGIN */
struct list_head *list;

list = &(pool->partial_slab_list);
if (list_empty(list)){
        pool->current_slab = NULL;
}
else{
        struct slab_header *slab;

        slab = (struct slab_header *)list_entry(
                list->next, struct slab_header, node);
        pool->current_slab = slab;
        list_del(list->next);
}
/* BLANK END */
/* LAB 2 TODO 2 END */
```

该段代码从 partial_slab_list 中取出 partial slab 作为新的 slab。同时当 partial_slab_list 为空时，给 current_slab 标记为空，以便在 alloc_in_slab_impl 函数中读取标记并为它分配新的 current_slab。最后把获得的 slab 从 partial_slab_list 中删去。

在 alloc_in_slab_impl 函数中，填入以下代码：

```
/* LAB 2 TODO 2 BEGIN */
/*
 * Hint: Find a free slot from the free list of current slab.
 * If current slab is full, choose a new slab as the current one.
 */
/* BLANK BEGIN */
free_list = (struct slab_slot_list *)current_slab->free_list_head;
BUG_ON(free_list == NULL);

next_slot = free_list->next_free;
current_slab->free_list_head = next_slot;

current_slab->current_free_cnt -= 1;
if (unlikely(current_slab->current_free_cnt == 0))
        choose_new_current_slab(&slab_pool[order]);
/* BLANK END */
/* LAB 2 TODO 2 END */
```

该段代码首先遍历空闲链表拿到下一个空闲的 slot，然后更新对应的信息。若此时已经没有

空闲的 slot，则调用已经实现的 choose_new_current_slab 函数创建新的 slab。
在 free_in_slab 函数中填入以下代码:

```
/* LAB 2 TODO 2 BEGIN */
/*
 * Hint: Free an allocated slot and put it back to the free list.
 */
/* BLANK BEGIN */

slot->next_free = slab->free_list_head;
slab->free_list_head = (void *)slot;
slab->current_free_cnt += 1;

/* BLANK END */
/* LAB 2 TODO 2 END */
```

该段代码将待释放的 slot 放回 slab 并更新对应信息。

练习题 3：
在_kmalloc 函数中填入以下代码:

```
                /* LAB 2 TODO 3 BEGIN */
                /* Step 1: Allocate in slab for small requests. */
                /* BLANK BEGIN */
                addr = alloc_in_slab(size, real_size);

                /* BLANK END */
#if ENABLE_MEMORY_USAGE_COLLECTING == ON
                if(is_record && collecting_switch) {
                        record_mem_usage(*real_size, addr);
                }
#endif
        } else {
                /* Step 2: Allocate in buddy for large requests. */
                /* BLANK BEGIN */
                order = size_to_page_order(size);
                addr = _get_pages(order, is_record);
                /* BLANK END */
                /* LAB 2 TODO 3 END */
```

该函数在申请内存小时，调用 alloc_in_slab 函数使用 slab 分配小内存；在申请内存大时直接用伙伴系统分配对应大小的页。

练习题 4：
对于 query_in_pgtbl 函数，填入以下代码:

```c
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
ptp_t *phys_page;
pte_t *pte;
int ret;

l0_ptp = (ptp_t *)pgtbl;
ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false, NULL);
if (ret < 0){
        return ret;
}

ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false, NULL);
if (ret < 0){
        return ret;
}
else if (ret == BLOCK_PTP){
        *pa = (pte->l1_block.pfn << L1_INDEX_SHIFT) | (GET_VA_OFFSET_L1(va));
        if (entry != NULL){
                *entry = pte;
        }
        return 0;
}

ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false, NULL);
if (ret < 0){
        return ret;
}
else if (ret == BLOCK_PTP){
        *pa = (pte->l2_block.pfn << L2_INDEX_SHIFT) | (GET_VA_OFFSET_L2(va));
        if (entry != NULL){
                *entry = pte;
        }
        return 0;
}

ret = get_next_ptp(l3_ptp, L3, va, &phys_page, &pte, false, NULL);
if (ret < 0){
        return ret;
}
*pa = (pte->l3_page.pfn << L3_INDEX_SHIFT) | (GET_VA_OFFSET_L3(va));
if (entry != NULL){
        *entry = pte;
}
```

对于 map_range_in_pgtbl_common 函数，填入以下代码:

```c
s64 total_page_cnt;
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
pte_t *pte;
int ret;
int pte_index;
int i;

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

l0_ptp = (ptp_t *)pgtbl;

l1_ptp = NULL;
l2_ptp = NULL;
l3_ptp = NULL;

while(total_page_cnt > 0){
        ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, true, rss);
        BUG_ON(ret != 0);

        ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, true, rss);
        BUG_ON(ret != 0);

        ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, true, rss);
        BUG_ON(ret != 0);

        pte_index = GET_L3_INDEX(va);
        for (i = pte_index; i < PTP_ENTRIES; i++){
                pte_t new_pte_val;

                new_pte_val.pte = 0;
                new_pte_val.l3_page.is_valid = 1;
                new_pte_val.l3_page.is_page = 1;
                new_pte_val.l3_page.pfn = pa >> PAGE_SHIFT;
                set_pte_flags(&new_pte_val, flags, kind);
                l3_ptp->ent[i].pte = new_pte_val.pte;

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

对于 unmap_range_in_pgtbl 函数，填入以下代码:

```
u64 cnt = DIV_ROUND_UP(len, PAGE_SIZE);
u64 idx = 0;
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
ptp_t *phys_page;
pte_t *pte;
vaddr_t cur_va;
int ret;

while (idx < cnt){
        cur_va = va + idx * PAGE_SIZE;
        l0_ptp = (ptp_t *)pgtbl;
        ret = get_next_ptp(l0_ptp, L0, cur_va, &l1_ptp, &pte, false, rss);
        if (ret < 0){
                return ret;
        }
        ret = get_next_ptp(l1_ptp, L1, cur_va, &l2_ptp, &pte, false, rss);
        if (ret < 0){
                return ret;
        }
        else if (ret == BLOCK_PTP){
                idx += L1_PER_ENTRY_PAGES;
                pte->pte = PTE_DESCRIPTOR_INVALID;
                continue;
        }
        ret = get_next_ptp(l2_ptp, L2, cur_va, &l3_ptp, &pte, false, rss);
        if (ret < 0){
                return ret;
        }
        else if (ret == BLOCK_PTP){
                idx += L2_PER_ENTRY_PAGES;
                pte->pte = PTE_DESCRIPTOR_INVALID;
                continue;
        }
        ret = get_next_ptp(l3_ptp, L3, cur_va, &phys_page, &pte, false, rss);
        if (ret < 0){
                return ret;
        }
        else{
                idx += L3_PER_ENTRY_PAGES;
                pte->pte = PTE_DESCRIPTOR_INVALID;
                recycle_pgtable_entry(l0_ptp, l1_ptp, l2_ptp, l3_ptp, cur_va, rss);
        }
}
```

对于 mprotect_in_pgtbl 函数，填入以下代码：

```
u64 cnt = DIV_ROUND_UP(len, PAGE_SIZE);
u64 idx = 0;
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
ptp_t *phys_page;
pte_t *pte;
vaddr_t cur_va;
int ret;

while(idx < cnt){
        cur_va = va + idx * PAGE_SIZE;
        l0_ptp = (ptp_t *)pgtbl;
        ret = get_next_ptp(l0_ptp, L0, cur_va, &l1_ptp, &pte, false, NULL);
        if (ret < 0){
                return ret;
        }
        ret = get_next_ptp(l1_ptp, L1, cur_va, &l2_ptp, &pte, false, NULL);
        if (ret < 0){
                return ret;
        }
        else if (ret == BLOCK_PTP){
                idx += L1_PER_ENTRY_PAGES;
                continue;
        }
        ret = get_next_ptp(l2_ptp, L2, cur_va, &l3_ptp, &pte, false, NULL);
        if (ret < 0){
                return ret;
        }
        else if (ret == BLOCK_PTP){
                idx += L2_PER_ENTRY_PAGES;
                continue;
        }
        ret = get_next_ptp(l3_ptp, L3, cur_va, &phys_page, &pte, false, NULL);
        if (ret < 0){
                return ret;
        }
        else{
                idx += L3_PER_ENTRY_PAGES;
                set_pte_flags(pte, flags, USER_PTE);
        }
}
```

对于以上 4 个函数，按提示依次遍历各级页表，进行对应的操作(查询、映射、取消映射、修改权限)即可。

思考题 5：

需要配置页表描述符的访问权限字段（Access Permission）为只读。在发生页错误后，首先会判断页错误发生的原因，若是由写实拷贝的权限问题引起的，则会重新为进程分配物理页，将共享的数据复制到该物理页中并设置权限为可读可写，再更新页表的映射关系。

思考题 6：

会产生内部碎片，分配的内存不会被充分利用，浪费了部分内存。同时粗粒度映射限制了内核对不同区域的控制，难以对内核内存进行严格的权限划分。

练习题 8：

在 do_page_fault 函数中填入以下代码：

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

该段代码使用已有的 handle_trans_fault 函数处理翻译错误，并将返回值赋给 ret。

练习题 9：

在 find_vmr_for_va 函数中填入以下代码：

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

代码利用 rb_search 函数找到虚拟地址的 node，再使用 rb_entry 得到对应的 vmr。

练习题 10：

在 handle_trans_fault 函数中填入以下代码：

```
/* LAB 2 TODO 7 BEGIN */
/* BLANK BEGIN */
/* Hint: Allocate a physical page and clear it to 0. */
void *va = get_pages(0);
pa = virt_to_phys(va);
memset(va, 0, PAGE_SIZE);
/* BLANK END */
/*
 * Record the physical page in the radix tree:
 * the offset is used as index in the radix tree
 */
```

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

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

根据提示，首先调用 get_pages 分配物理页并将内容初始化为 0。随后调用 map_range_in_pgtbl 函数配置页表映射。

运行 make qemu，可以看到成功进入了 Shell: