实验二：内存管理

练习题1：

> 练习题 1：完成 `kernel/mm/buddy.c` 中的 `split_chunk`、`merge_chunk`、`buddy_get_pages`、 和 `buddy_free_pages` 函数中的 `LAB 2 TODO 1` 部分，其中 `buddy_get_pages` 用于分配指定阶大小的连续物理页，`buddy_free_pages` 用于释放已分配的连续物理页。
>
> 提示：
>
> - 可以使用 `kernel/include/common/list.h` 中提供的链表相关函数和宏如 `init_list_head`、`list_add`、`list_del`、`list_entry` 来对伙伴系统中的空闲链表进行操作
> - 可使用 `get_buddy_chunk` 函数获得某个物理内存块的伙伴块
> - 更多提示见代码注释

- `split_chunk` 代码如下：

```
static struct page *split_chunk(struct phys_mem_pool *pool, int order,
                                struct page *chunk)
{
        /* 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;
        }

        // split the chunk to two parts
        chunk->order--;
        struct page *buddy_chunk = get_buddy_chunk(pool, chunk);
        buddy_chunk->order = chunk->order;

        list_add(&buddy_chunk->node,
                 &(pool->free_lists[chunk->order].free_list));
        pool->free_lists[chunk->order].nr_free += 1;

        buddy_chunk->pool = chunk->pool;
        buddy_chunk->allocated = 0;

        return split_chunk(pool, order, chunk);

        /* BLANK END */
        /* LAB 2 TODO 1 END */
}
```

如果当前chunk的order就是我们需要的order直接返回，如果不是则需要split。

先将order-1，调用 `get_buddy_chunk` 获取 `buddy_chunk` 同时为其属性复制，再递归调用 `split_chunk`。

- `merge_chunk` 代码如下：

```
static struct page *merge_chunk(struct phys_mem_pool *pool, struct page
*chunk)
{
        /* LAB 2 TODO 1 BEGIN */
        /*
         * Hint: Recursively merge current chunk with its buddy
         * if possible.
         */
        /* BLANK BEGIN */
        if (chunk->order == (BUDDY_MAX_ORDER - 1)) {
                return chunk;
        }

        struct page *buddy_chunk = get_buddy_chunk(pool, chunk);
        if (buddy_chunk == NULL || buddy_chunk->order != chunk->order) {
                return chunk;
        }
        if (buddy_chunk->allocated == 1) {
                return chunk;
        }

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

        buddy_chunk->order++;
        chunk->order++;

        if (chunk > buddy_chunk) {
                return merge_chunk(pool, buddy_chunk);
        } else {
                return merge_chunk(pool, chunk);
        }
        /* BLANK END */
        /* LAB 2 TODO 1 END */
}
```

当 `chunk` 达到最大，或者 `buddy_chunk` 不存在，已被分配，不是整块，就不能 `merge` 直接返回 如果能够 `merge`，则把 `buddy_chunk` 从对应的 `free_list` 删去，设置对应 `order`，并选取与 `chunk` 对 应更前的地址，作为新的 `chunk` 地址。

- `buddy_get_page` 补全代码如下：

```
        cur_order = order;
        for (; cur_order < BUDDY_MAX_ORDER; cur_order++) {
                if (pool->free_lists[cur_order].nr_free > 0) {
                        break;
                }
        }
        // if there is no free chunk, return NULL
        if (cur_order == BUDDY_MAX_ORDER) {
```

```
                page = NULL;
        } else {
                free_list = pool->free_lists[cur_order].free_list.next;
                page = list_entry(free_list, struct page, node);
                pool->free_lists[page->order].nr_free--;
                page = split_chunk(pool, order, page);
                page->allocated = 1;
                list_del(&page->node);
        }
```

找到大于等于所需 `order` 的非空 `free_list`，从 `pool` 对该 `free_list` 的 `chunk` 进行 `split`，直到得到所需 `order` 的 `page`，然后从 `free_list` 中删去 `page`，并设置为已分配。

- `buddy_free_pages` 代码如下：

```
void buddy_free_pages(struct phys_mem_pool *pool, struct page *page)
{
        int order;
        struct list_head *free_list;

        lock(&pool->buddy_lock);

        /* 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++;
        /* BLANK END */
        /* LAB 2 TODO 1 END */

        unlock(&pool->buddy_lock);
}
```

将对应 `page` 设置为非分配，进行 `merge` 之后，放到对应的 `free_list` 中。

练习题2：

> 练习题 2：完成 `kernel/mm/slab.c` 中的 `choose_new_current_slab`、`alloc_in_slab_impl` 和 `free_in_slab` 函数中的 `LAB 2 TODO 2` 部分，其中 `alloc_in_slab_impl` 用于在 slab 分配器中分配指定阶大小的内存，而 `free_in_slab` 则用于释放上述已分配的内存。
>
> 提示：
>
> - 你仍然可以使用上个练习中提到的链表相关函数和宏来对 SLAB 分配器中的链表进行操作
> - 更多提示见代码注释

- `choose_new_current_slab` 代码如下：

```
static void choose_new_current_slab(struct slab_pointer *pool)
{
        /* 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 */
}
```

利用`pool->partial_slab_list`获得对应的`partial slab`链表，如果该链表为空，则返回`NULL`，获得新的`current slab`失败，如果非空，则利用`list_entry`方法获得对应的`slab`的地址。然后赋值给`pool`的`current_slab`，最后把该`slab`从`partial slab`链上去。

- `alloc_in_slab_impl`代码如下：

```

        /* 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--;

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

根据提示，利用`current_slab`的`free_list_head`属性获取对应`slot`的`free_list`链表 然后通过操作指针从链表中获取`slot`，把他从链表中删去。如果`current_slab`的中没有空余`slot`，再选择新的`current slab`。

- `free_in_slab`代码如下：

```
        /* 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 = slot;
        slab->current_free_cnt++;
        /* BLANK END */
        /* LAB 2 TODO 2 END */
```

只要将`alloct`的过程逆向即可

练习题3：

> 练习题 3：完成 `kernel/mm/kmalloc.c` 中的 `_kmalloc` 函数中的 `LAB 2 TODO 3` 部分，在适当位置调用对应的函数，实现 `kmalloc` 功能
>
> 提示：
>
> - 你可以使用 `get_pages` 函数从伙伴系统中分配内存，使用 `alloc_in_slab` 从 SLAB 分配器中分配内存
> - 更多提示见代码注释

- `_kmalloc`代码如下：

```
void *_kmalloc(size_t size, bool is_record, size_t *real_size)
{
        void *addr;
        int order;

        if (unlikely(size == 0))
                return ZERO_SIZE_PTR;

        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 */
#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 */
        }

        BUG_ON(!addr);
        return addr;
}
```

调用已有的`alloc_in_slab`分配小块的内存，调用`get_pages`分配大块内存即可。

练习题4：

> 练习题 4：完成`kernel/arch/aarch64/mm/page_table.c`中的`query_in_pgtbl`、`map_range_in_pgtbl_common`、`unmap_range_in_pgtbl`和`mprotect_in_pgtbl`函数中的`LAB 2 TODO 4`部分，分别实现页表查询、映射、取消映射和修改页表权限的操作，以 4KB 页为粒度。
>
> 提示：
>
> - 需要实现的函数内部无需刷新 TLB，TLB 刷新会在这些函数的外部进行
> - 实现中可以使用`get_next_ptp`、`set_pte_flags`、`virt_to_phys`、`GET_LX_INDEX`等已经给定的函数和宏
> - 更多提示见代码注释

- `query_in_pgtbl`代码如下：

```
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
{
        /* LAB 2 TODO 4 BEGIN */
        /*
         * Hint: Walk through each level of page table using `get_next_ptp`,
         * return the pa and pte until a L0/L1 block or page, return
         * `-ENOMAPPING` if the va is not mapped.
         */
        /* BLANK BEGIN */
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp, *ptp;
        pte_t *pte;
        int ret;

        l0_ptp = (ptp_t *)pgtbl;
        l1_ptp = NULL;
        l2_ptp = NULL;
        l3_ptp = NULL;

        ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false);
        if (ret == -ENOMAPPING) {
                return ret;
        }

        ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false);
        if (ret == -ENOMAPPING) {
                return -ENOMAPPING;
```

```
        } else if (ret == BLOCK_PTP) {
                if (entry != NULL) {
                        *entry = pte;
                }
                *pa = virt_to_phys((vaddr_t)l2_ptp) +
GET_VA_OFFSET_L1(va);
                return 0;
        }

        ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false);
        if (ret == -ENOMAPPING) {
                return ret;
        } else if (ret == BLOCK_PTP) {
                if (entry != NULL) {
                        *entry = pte;
                }
                *pa = virt_to_phys((vaddr_t)l3_ptp) +
GET_VA_OFFSET_L2(va);
                return 0;
        }

        ret = get_next_ptp(l3_ptp, L3, va, &ptp, &pte, false);

        if (ret == -ENOMAPPING) {
                return ret;
        }

        if (entry != NULL) {
                *entry = pte;
        }
        *pa = virt_to_phys(ptp) + GET_VA_OFFSET_L3(va);
        /* BLANK END */
        /* LAB 2 TODO 4 END */
        return 0;
}
```

利用`get_next_ptp`依次遍历对应`va`的每级页表，利用ret判断为block则返回对应页表项和物理页。

如果未分配，则返回`-ENOMAPPING`；如果遍历到物理页且分配， 则返回对应页表项和物理页。

- `map_range_in_pgtbl_common`代码如下：

```
static int map_range_in_pgtbl_common(void *pgtbl, vaddr_t va, paddr_t
pa,
                                     size_t len, vmr_prop_t flags, int
kind)
{
        /* LAB 2 TODO 4 BEGIN */
        /*
         * Hint: Walk through each level of page table using
`get_next_ptp`,
         * create new page table page if necessary, fill in the final
level
         * pte with the help of `set_pte_flags`. Iterate until all pages
are
```

```
         * mapped.
         * Since we are adding new mappings, there is no need to flush TLBs.
         * Return 0 on success.
         */
        /* BLANK BEGIN */
        u64 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);
                BUG_ON(ret != 0);

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

                ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, true);
                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;

                        total_page_cnt -= 1;
                        if (total_page_cnt == 0)
                                break;
                }
        }
        /* BLANK END */
        /* LAB 2 TODO 4 END */
        return 0;
```

```
}
```

利用`total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1 : 0);`计算总的需要映射的物理页数目，然后利用`get_next_ptp`找到对应的`va`的第三季页表， 并分配可能没有分配的页表页，最后遍历最后一级页表，依次分配物理页，直到达到对应的数目。

- `unmap_range_in_pgtbl`代码如下：

```
int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len)
{
        /* LAB 2 TODO 4 BEGIN */
        /*
         * Hint: Walk through each level of page table using `get_next_ptp`,
         * mark the final level pte as invalid. Iterate until all pages are
         * unmapped.
         * You don't need to flush tlb here since tlb is now flushed after
         * this function is called.
         * Return 0 on success.
         */
        /* BLANK BEGIN */
        u64 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);

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

        while (total_page_cnt > 0) {
                l0_ptp = (ptp_t *)pgtbl;
                ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L0_PER_ENTRY_PAGES;
                        va += L0_PER_ENTRY_PAGES * PAGE_SIZE;
                        continue;
                }

                ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L1_PER_ENTRY_PAGES;
                        va += L1_PER_ENTRY_PAGES * PAGE_SIZE;
```

```
                        continue;
                } else if (ret == BLOCK_PTP) {
                        pte->pte = PTE_DESCRIPTOR_INVALID;
                        va += PAGE_SIZE * PTP_ENTRIES * PTP_ENTRIES;

                        total_page_cnt -= PTP_ENTRIES * PTP_ENTRIES;
                        continue;
                }

                ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte,
false);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L2_PER_ENTRY_PAGES;
                        va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
                        continue;
                } else if (ret == BLOCK_PTP) {
                        pte->pte = PTE_DESCRIPTOR_INVALID;
                        va += PAGE_SIZE * PTP_ENTRIES;

                        total_page_cnt -= PTP_ENTRIES;
                        continue;
                }

                pte_index = GET_L3_INDEX(va);
                for (i = pte_index; i < PTP_ENTRIES; ++i) {
                        l3_ptp->ent[i].pte = PTE_DESCRIPTOR_INVALID;

                        va += PAGE_SIZE;

                        total_page_cnt -= 1;
                        if (total_page_cnt == 0)
                                break;
                }
        }
        /* BLANK END */
        /* LAB 2 TODO 4 END */

        dsb(ishst);
        isb();

        return 0;
}
```

在调用`get_next_ptp`时，将`alloct`设为`false`，并以此判断对应的页表项是块描述符还是页 描述符，还是指向下一级页表的基地址。如果是块描述符，直接`unmap`(`unmap`操作通过对`pte`页表项赋值为 0 实现)后，需要对`va`加上对应块的大小，并对计数用的`total_page_cnt`减掉对应的物理页数；如果是页描述符，`unmap`后，需要对`va`加上对应页的大小，并对计数用的`total_page_cnt`减1；如果未映射，则省去`unmap`操作，直接对`va`、`total_page_cnt`做对应操作。

- `mprotect_in_pgtbl`代码如下：

```
int mprotect_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, vmr_prop_t
flags)
{
```

```
        /* LAB 2 TODO 4 BEGIN */
        /*
         * Hint: Walk through each level of page table using `get_next_ptp`,
         * modify the permission in the final level pte using `set_pte_flags`.
         * The `kind` argument of `set_pte_flags` should always be `USER_PTE`.
         * Return 0 on success.
         */
        /* BLANK BEGIN */
        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, false);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L0_PER_ENTRY_PAGES;
                        va += L0_PER_ENTRY_PAGES * PAGE_SIZE;
                        continue;
                }

                ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L1_PER_ENTRY_PAGES;
                        va += L1_PER_ENTRY_PAGES * PAGE_SIZE;
                        continue;
                } else if (ret == BLOCK_PTP) {
                        set_pte_flags(pte, flags, USER_PTE);
                        va += PAGE_SIZE * PTP_ENTRIES * PTP_ENTRIES;

                        total_page_cnt -= PTP_ENTRIES * PTP_ENTRIES;
                        continue;
                }

                ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L2_PER_ENTRY_PAGES;
```

```
                        va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
                        continue;
                } else if (ret == BLOCK_PTP) {
                        set_pte_flags(pte, flags, USER_PTE);
                        va += PAGE_SIZE * PTP_ENTRIES;

                        total_page_cnt -= PTP_ENTRIES;
                        continue;
                }

                pte_index = GET_L3_INDEX(va);
                for (i = pte_index; i < PTP_ENTRIES; ++i) {
                        set_pte_flags(&l3_ptp->ent[i], flags, USER_PTE);

                        va += PAGE_SIZE;

                        total_page_cnt -= 1;
                        if (total_page_cnt == 0)
                                break;
                }
        }
        /* BLANK END */
        /* LAB 2 TODO 4 END */
        return 0;
}
```

基本思路和`unmap_range_in_pgtbl`一致，只是将`unmap`操作变成`set_pte_flags(&l3_ptp->ent[i] ,flags,USER_PTE);`

思考题5：

> 思考题 5：阅读 Arm Architecture Reference Manual，思考要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的哪个/哪些字段，并在发生页错误时如何处理。（在完成第三部分后，你也可以阅读页错误处理的相关代码，观察 ChCore 是如何支持 Cow 的）

- 为支持写时拷贝需要配置：
  - **Access Permissions (AP)**：页表描述符中的访问权限字段可以用于控制页面的读写权限，对于共享的页面，一般设置为只读，防止不同进程修改页内容。
  - **Domain Access Control (DACR)**：DACR字段用于定义访问控制域，用于控制哪些进程可以访问特定的内存区域。
- 发生页错误时的处理：
  - 引到对应的物理页，将物理页的内容拷贝到另一块物理页中。
  - 然后将该物理页映射到对应的虚拟地址，并在页表项中填写好对应的AP为可读可写。
  - 然后在触发异常的地址继续执行程序。

思考题6：

> 思考题 6：为了简单起见，在 ChCore 实验 Lab1 中没有为内核页表使用细粒度的映射，而是直接沿用了启动时的粗粒度页表，请思考这样做有什么问题。

- 被映射的物理页可能不能被充分利用，存在较大的内部碎片。

练习题8：

> 练习题 8: 完成 `kernel/arch/aarch64/irq/pgfault.c` 中的 `do_page_fault` 函数中的 `LAB 2 TODO 5` 部分，将缺页异常转发给 `handle_trans_fault` 函数。

- `do_page_fault` 代码如下：

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

调用 `ret = handle_trans_fault(current_thread->vmspace, fault_addr);` 即可

练习题9：

> 练习题 9: 完成 `kernel/mm/vmspace.c` 中的 `find_vmr_for_va` 函数中的 `LAB 2 TODO 6` 部分，找到一个虚拟地址找在其虚拟地址空间中的 VMR。

- `find_vmr_for_va` 代码如下：

```
struct vmregion *find_vmr_for_va(struct vmspace *vmspace, vaddr_t addr)
{
        /* 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), addr, cmp_vmr_and_va);

        if (node == NULL)
                return node;

        return rb_entry(node, struct vmregion, tree_node);
        /* BLANK END */
        /* LAB 2 TODO 6 END */
}
```

调用宏函数 `rb_search`，`rb_entry` 即可找到对应的`vmr`。

练习题10：

> 练习题 10: 完成 `kernel/mm/pgfault_handler.c` 中的 `handle_trans_fault` 函数中的 `LAB 2 TODO 7` 部分（函数内共有 3 处填空，不要遗漏），实现 `PMO_SHM` 和 `PMO_ANONYM` 的按需物理页分配。你可以阅读代码注释，调用你之前见到过的相关函数来实现功能。

- `handle_trans_fault` 代码如下：

```
                if (pa == 0) {
                        /*
                         * Not committed before. Then, allocate the
physical
                         * page.
                         */
```

```
                        /* LAB 2 TODO 7 BEGIN */
                        /* BLANK BEGIN */
                        /* Hint: Allocate a physical page and clear it
to 0. */
                        pa = virt_to_phys(get_pages(0));
                        memset(phys_to_virt(pa), 0, PAGE_SIZE);
                        /* BLANK END */
                        /*
                         * Record the physical page in the radix tree:
                         * the offset is used as index in the radix tree
                         */
                        kdebug("commit: index: %ld, 0x%lx\n", index,
pa);
                        commit_page_to_pmo(pmo, index, pa);

                        /* Add mapping in the page table */
                        lock(&vmspace->pgtbl_lock);
                        /* BLANK BEGIN */
                        map_range_in_pgtbl(vmspace->pgtbl,
                                           fault_addr,
                                           pa,
                                           PAGE_SIZE,
                                           perm);
                        /* BLANK END */
                        unlock(&vmspace->pgtbl_lock);
                } else {
                        /*
                         * pa != 0: the faulting address has be
committed a
                         * physical page.
                         *
                         * For concurrent page faults:
                         *
                         * When type is PMO_ANONYM, the later faulting
threads
                         * of the process do not need to modify the page
                         * table because a previous faulting thread will
do
                         * that. (This is always true for the same
process)
                         * However, if one process map an anonymous pmo
for
                         * another process (e.g., main stack pmo), the
faulting
                         * thread (e.g, in the new process) needs to
update its
                         * page table.
                         * So, for simplicity, we just update the page
table.
                         * Note that adding the same mapping is
harmless.
                         *
                         * When type is PMO_SHM, the later faulting
threads
                         * needs to add the mapping in the page table.
                         * Repeated mapping operations are harmless.
```

```
                         */
                        if (pmo->type == PMO_SHM || pmo->type ==
PMO_ANONYM) {
                                /* Add mapping in the page table */
                                lock(&vmspace->pgtbl_lock);
                                /* BLANK BEGIN */
                                map_range_in_pgtbl(vmspace->pgtbl,
                                                   fault_addr,
                                                   pa,
                                                   PAGE_SIZE,
                                                   perm);
                                /* BLANK END */
                                /* LAB 2 TODO 7 END */
                                unlock(&vmspace->pgtbl_lock);
                        }
                }
```

利用 `virt_to_phys(get_pages(0));` 分配物理页 。

利用 `memset(phys_to_virt(pa), 0 , PAGE_SIZE);` 清空物理页 。

利用 `map_range_in_pgtbl(vmspace->pgtbl,fault_addr,pa,PAGE_SIZE,perm,&rss);` 对物理页进行映射。