Report of Lab2

王焕宇 522030910212

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。

```c
__maybe_unused static struct page *split_chunk(struct phys_mem_pool *__maybe_unused pool,
                                                int __maybe_unused order,
                                                struct page *__maybe_unused chunk)
{
        while (chunk->order > order){ // not reaching the aimed order
                struct page *buddy;
                chunk->order -= 1; // decrease the order
                buddy = get_buddy_chunk(pool, chunk);
                buddy->order = chunk->order;
                buddy->allocated = 0;
                pool->free_lists[buddy->order].nr_free += 1; // increase the number of free
                list_add(&(buddy->node), &(pool->free_lists[buddy->order].free_list)); // add the buddy into free list
        }
        BUG_ON(chunk == NULL);
        return chunk;
}

__maybe_unused static struct page * merge_chunk(struct phys_mem_pool *__maybe_unused pool,
                                                struct page *__maybe_unused chunk)
{
        struct page *buddy_chunk;
        // the chunk has already been the largest one
        if (chunk->order == (BUDDY_MAX_ORDER - 1)) {
                return chunk;
        }
        // locate the buddy_chunk of chunk
        buddy_chunk = get_buddy_chunk(pool, chunk);
        // if the buddy_chunk does not exist, no further merge is required
        if (buddy_chunk == NULL) {
                return chunk;
        }
        if (buddy_chunk->allocated == 1) {
                return chunk;
        }
        // the buddy_chunk is not free as a whole, no further merge is required
        if (buddy_chunk->order != chunk->order) {
                return chunk;
        }
        // remove the buddy_chunk from its current free list
        list_del(&(buddy_chunk->node));
        pool->free_lists[buddy_chunk->order].nr_free -= 1;
        // merge the two buddies and get a large chunk
        buddy_chunk->order += 1;
        chunk->order += 1;
        if (chunk > buddy_chunk) {
                chunk = buddy_chunk;
        }
        // keep merging
        return merge_chunk(pool, chunk);
}

struct page *buddy_get_pages(struct phys_mem_pool *pool, int order)
{
        int cur_order;
        struct list_head *free_list;
        struct page *page = NULL;

        if (unlikely(order >= BUDDY_MAX_ORDER)) {
                kwarn("ChCore does not support allocating such too large "
                      "contious physical memory\n");
                return NULL;
        }

        lock(&pool->buddy_lock);

        for (cur_order = order; cur_order < BUDDY_MAX_ORDER; cur_order++) {
```

```
                free_list = &(pool->free_lists[cur_order].free_list);
                if(!list_empty(free_list)) {
                        struct list_head* prepare = free_list->next;
                        page = list_entry(prepare, struct page, node); // get the addr of the page which has free_list's addr as node
                        list_del(&(page->node));
                        pool->free_lists[cur_order].nr_free -= 1;
                        page = split_chunk(pool, order, page);
                        page->allocated = 1;
                        page->order = order;
                        break;
                }
        }
out: __maybe_unused
        unlock(&pool->buddy_lock);
        return page;
}

void buddy_free_pages(struct phys_mem_pool *pool, struct page *page)
{
        int order;
        struct list_head *free_list;
        lock(&pool->buddy_lock);
        // mark the chunk page as free
        page->allocated = 0;
        // merge the freed chunk
        page = merge_chunk(pool, page);
        // put the merged chunk into its corresponding free list
        order = page->order;
        free_list = &(pool->free_lists[order].free_list);
        list_add(&page->node, free_list);
        pool->free_lists[order].nr_free += 1;
        unlock(&pool->buddy_lock);
}
```

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。

```
static void choose_new_current_slab(struct slab_pointer * __maybe_unused pool)
{
        struct list_head *list;
        list = &(pool->partial_slab_list);
        if (list_empty(list)){
                pool->current_slab = NULL; // no available slab
        } else {
                struct slab_header *slab;
                // choose the first in the partial_slab_list
                slab = (struct slab_header *)list_entry(list->next, struct slab_header, node);
                pool->current_slab = slab;
                list_del(list->next);
        }
        return;
}

static void *alloc_in_slab_impl(int order)
{
        struct slab_header *current_slab;
        struct slab_slot_list *free_list;
        void *next_slot;
        UNUSED(next_slot);

        lock(&slabs_locks[order]);

        current_slab = slab_pool[order].current_slab;
        /* When serving the first allocation request. */
        if (unlikely(current_slab == NULL)) {
                current_slab = init_slab_cache(order, SIZE_OF_ONE_SLAB);
                if (current_slab == NULL) {
                        unlock(&slabs_locks[order]);
                        return NULL;
                }
                slab_pool[order].current_slab = current_slab;
        }

        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;
        // when current_slab is full, choose a new slab as the current one
        if (unlikely(current_slab->current_free_cnt == 0)) choose_new_current_slab(&slab_pool[order]);

        unlock(&slabs_locks[order]);

        return (void *)free_list;
}

void free_in_slab(void *addr)
{
        struct page *page;
        struct slab_header *slab;
        struct slab_slot_list *slot;
        int order;

        slot = (struct slab_slot_list *)addr;
        page = virt_to_page(addr);
        if (!page) {
                kdebug("invalid page in %s", __func__);
                return;
        }
```

```
        slab = page->slab;
        order = slab->order;
        lock(&slabs_locks[order]);

        try_insert_full_slab_to_partial(slab);

#if ENABLE_DETECTING_DOUBLE_FREE_IN_SLAB == ON
        /*
         * SLAB double free detection: check whether the slot to free is
         * already in the free list.
         */
        if (check_slot_is_free(slab, slot) == 1) {
                kinfo("SLAB: double free detected. Address is %p\n",
                      (unsigned long)slot);
                BUG_ON(1);
        }
#endif
        slot->next_free = slab->free_list_head;
        slab->free_list_head = slot;
        slab->current_free_cnt += 1; // free the slot

        try_return_slab_to_buddy(slab, order);

        unlock(&slabs_locks[order]);
}
```

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

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

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

        if (size <= SLAB_MAX_SIZE) {
                addr = alloc_in_slab(size, real_size);
                if (real_size) {
                        *real_size = size;
                }
#if ENABLE_MEMORY_USAGE_COLLECTING == ON
                if(is_record && collecting_switch) {
                        record_mem_usage(*real_size, addr);
                }
#endif
        } else {
                order = size_to_page_order(size);
                addr = get_pages(order);
                if (real_size != NULL) {
                        *real_size = (BUDDY_PAGE_SIZE << order);
                }
        }

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

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。

```
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
{
        ptp_t *cur_ptp = (ptp_t *)pgtbl;
        for(int level = L0; level <= L3; level++) {
                ptp_t *next_ptp;
                pte_t *cur_entry;
                if(get_next_ptp(cur_ptp, level, va, &next_ptp, &cur_entry, false, NULL) == -ENOMAPPING) return -ENOMAPPING;
                if(level == L3) {
                        *pa = GET_PADDR_IN_PTE(cur_entry) + GET_VA_OFFSET_L3(va);
                        if(entry != NULL) *entry = cur_entry;
                        break;
                }
                cur_ptp = next_ptp;
        }
        return 0;
}

static int map_range_in_pgtbl_common(void *pgtbl, vaddr_t va, paddr_t pa,
                                     size_t len, vmr_prop_t flags, int kind,
                                     __maybe_unused long *rss)
{
        s64 total_page_cnt;
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
        pte_t *pte;
        int ret;
        int pte_index; // the index of pte in the last level pg
        int i;

        BUG_ON(pgtbl == NULL);
        BUG_ON(va % PAGE_SIZE);
        total_page_cnt = len/PAGE_SIZE + (len % PAGE_SIZE > 0);
        l0_ptp = (ptp_t *)pgtbl;
        l1_ptp = NULL;
        l2_ptp = NULL;
        l3_ptp = NULL;
        while (total_page_cnt > 0) {
                // l0
                ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, true, rss);
                BUG_ON(ret != 0);
                // l1
                ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, true, rss);
                BUG_ON(ret != 0);
                // l2
                ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, true, rss);
                BUG_ON(ret != 0);
                // l3
                // get the index of pte
                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;
                }
        }
        dsb(ishst);
        isb();
        return 0;
```

```
}

int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len,
                         __maybe_unused long *rss)
{
        s64 total_page_cnt;
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp, *current_ptp;
        pte_t *pte;
        int ret;
        int pte_index;
        int i;

        total_page_cnt = len/PAGE_SIZE + (len % PAGE_SIZE > 0);
        current_ptp = (ptp_t *)pgtbl;
        l0_ptp = (ptp_t *)pgtbl;
        l1_ptp = NULL;
        l2_ptp = NULL;
        l3_ptp = NULL;

        while(total_page_cnt > 0) {
                // L0
                ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false, rss);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L0_PER_ENTRY_PAGES;
                        continue;
                }
                // L1
                ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false, rss);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L1_PER_ENTRY_PAGES;
                        continue;
                }
                // L2
                ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false, rss);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L2_PER_ENTRY_PAGES;
                        continue;
                }
                // L3
                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;
                }

        }
        dsb(ishst);
        isb();

        return 0;
}

int mprotect_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, vmr_prop_t flags)
{
        s64 total_page_cnt;
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp, *current_ptp;
        pte_t *pte;
        int ret;
        int pte_index;
        int i;

        total_page_cnt = len/PAGE_SIZE + (len % PAGE_SIZE > 0);
        current_ptp = (ptp_t *)pgtbl;
        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, NULL);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L0_PER_ENTRY_PAGES;
                        continue;
                }
                ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false, NULL);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L1_PER_ENTRY_PAGES;
                        continue;
                }
                ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false, NULL);
                if (ret == -ENOMAPPING) {
                        total_page_cnt -= L2_PER_ENTRY_PAGES;
                        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;
                        }
                }

         }
        return 0;
}
```

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）1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）

L3页表项中的AP字段用于定义物理页的读写权限。实现写时拷贝机制时，该物理页应设为可读不可写，因此AP字段应设置为11。当有应用程序试图对这个物理页进行写操作时，会引发一个访问权限异常。随后，操作系统会复制这个物理页，将复制页的AP字段设置为可读可写(01)，更新相应的页表项，然后让应用程序继续执行写操作。

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。

粗粒度页表映射可能会产生很多内部碎片

挑战题 7

使用前面实现的 page_table.c 中的函数，在内核启动后的 main 函数中重新配置内核页表，进行细粒度的映射。

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。

仅需添加一行

```
handle_trans_fault(current_thread->vmspace, fault_addr);
```

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。

```
__maybe_unused struct vmregion *find_vmr_for_va(struct vmspace *vmspace,
                                                vaddr_t addr)
{
        struct vmregion *vmr;
        struct rb_node* rb_node;
        struct rb_root* root;
        root = &(vmspace->vmr_tree);
        rb_node = rb_search(root,addr, cmp_vmr_and_va); // search the vmr that includes this va
        vmr = rb_entry(rb_node, struct vmregion, tree_node); // get the vmr that has node as its `tree_node`
        return vmr;
}
```

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。

```c
int handle_trans_fault(struct vmspace *vmspace, vaddr_t fault_addr)
{
        struct vmregion *vmr;
        struct pmobject *pmo;
        paddr_t pa;
        unsigned long offset;
        unsigned long index;
        int ret = 0;

        lock(&vmspace->vmspace_lock);
        vmr = find_vmr_for_va(vmspace, fault_addr);

        if (vmr == NULL) {
                kinfo("handle_trans_fault: no vmr found for va 0x%lx!\n",
                      fault_addr);
                dump_pgfault_error();
                unlock(&vmspace->vmspace_lock);

#if defined(CHCORE_ARCH_AARCH64) || defined(CHCORE_ARCH_SPARC)
                return -EFAULT;
#endif
                sys_exit_group(-1);

                BUG("should not reach here");
        }

        pmo = vmr->pmo;
        offset = ROUND_DOWN(fault_addr, PAGE_SIZE) - vmr->start + vmr->offset;
        vmr_prop_t perm = vmr->perm;
        switch (pmo->type) {
        case PMO_ANONYM:
        case PMO_SHM: {
                BUG_ON(offset >= pmo->size);
                index = offset / PAGE_SIZE;

                fault_addr = ROUND_DOWN(fault_addr, PAGE_SIZE);

                pa = get_page_from_pmo(pmo, index);
                if (pa == 0) {
                        pa = virt_to_phys(get_pages(0)); // allocate a physical page
                        memset((void *)phys_to_virt(pa), 0, PAGE_SIZE); // clear the page
                        kdebug("commit: index: %ld, 0x%lx\n", index, pa);
                        commit_page_to_pmo(pmo, index, pa);

                        lock(&vmspace->pgtbl_lock);
                        long rss = 0;
                        map_range_in_pgtbl(vmspace->pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss);
                        vmspace->rss += rss;
                        unlock(&vmspace->pgtbl_lock);
                } else {
                        if (pmo->type == PMO_SHM || pmo->type == PMO_ANONYM) {
                                lock(&vmspace->pgtbl_lock);
                                long rss = 0;
                                map_range_in_pgtbl(vmspace->pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss);
                                vmspace->rss += rss;
                                unlock(&vmspace->pgtbl_lock);
                        }
                }

                if (perm & VMR_EXEC) {
                        arch_flush_cache(fault_addr, PAGE_SIZE, SYNC_IDCACHE);
                }

                break;
        }
        case PMO_FILE: {
```

```
        unlock(&vmspace->vmspace_lock);
        fault_addr = ROUND_DOWN(fault_addr, PAGE_SIZE);
        handle_user_fault(pmo, ROUND_DOWN(fault_addr, PAGE_SIZE));
        BUG("Should never be here!\n");
        break;
    }
    case PMO_FORBID: {
        kinfo("Forbidden memory access (pmo->type is PMO_FORBID).\n");
        dump_pgfault_error();

        unlock(&vmspace->vmspace_lock);
        sys_exit_group(-1);

        BUG("should not reach here");
        break;
    }
    default: {
        kinfo("handle_trans_fault: faulting vmr->pmo->type"
              "(pmo type %d at 0x%lx)\n",
              vmr->pmo->type,
              fault_addr);
        dump_pgfault_error();

        unlock(&vmspace->vmspace_lock);
        sys_exit_group(-1);

        BUG("should not reach here");
        break;
    }
    }

    unlock(&vmspace->vmspace_lock);
    return ret;
}
```

挑战题 11

我们在map_range_in_pgtbl_common、unmap_range_in_pgtbl 函数中预留了没有被使用过的参数rss 用来来统计map映射中实际的物理内存使用量1， 你需要修改相关的代码来通过Compute physical memory测试，不实现该挑战题并不影响其他部分功能的实现及测试。如果你想检测是否通过此部分测试，需要修改kernel/config.cmake中CHCORE_KERNEL_PM_USAGE_TEST为ON