以下给出了每个练习题对应的代码部分和相应的注释，其中一些注释是在报告中添加的以更好地解释实现过程。

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
static struct page *split_chunk(struct phys_mem_pool *pool, int order,
                                struct page *chunk)
{
  while (chunk -> order > order){ // Before reaching to the aimed oerder
    struct page *buddy;
    -- chunk -> order;  // decrease the order after spliting
    buddy = get_buddy_chunk(pool, chunk);
    buddy -> order = chunk -> order;
    buddy -> allocated = 0; // allocated is false
    ++ pool -> free_lists[buddy -> order].nr_free; // increase the number of free
    list_add(&(buddy -> node), &(pool -> free_lists[buddy -> order].free_list)); // add the buddy into free list of
corresponding order
  }
  return chunk;
}

static struct page *merge_chunk(struct phys_mem_pool *pool, struct page *chunk)
{
  while(chunk -> order < BUDDY_MAX_ORDER - 1){
    struct page *buddy = get_buddy_chunk(pool, chunk);
    if (buddy == NULL || buddy -> allocated || buddy -> order != chunk -> order){
      break;
    } // no buddy available or wrong buddy
    -- pool -> free_lists[chunk -> order].nr_free; // decrease free number of corresponding order after merge

    list_del(&(buddy -> node));

    if (chunk > buddy){
      chunk = buddy;
    } // make chunk the one with lower addr

    ++ chunk -> order;
    chunk -> allocated = 1;
  }
  return 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);

        /* 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.next;    /* first one is head */
      page = list_entry(free_list, struct page, node);    /* get the addr of the page which has free_list's addr as
node*/
      -- pool -> free_lists[page -> order].nr_free;
      list_del(&(page -> node));
      break;
    }
  }
  if(page == NULL) {
    unlock(&pool -> buddy_lock);
    return NULL;
  }
  page = split_chunk(pool, order, page);
  page -> allocated = 1;
        /* BLANK END */
out:
```

```c
	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);


	/* BLANK BEGIN */
	if (page == NULL || page -> allocated == 0) {return;}
	page -> allocated = 0; // free the page
	struct page *merged_chunk = merge_chunk(pool, page);
	merged_chunk -> allocated = 0;
	order = merged_chunk -> order;
	++ pool -> free_lists[order].nr_free;
	free_list = &(pool -> free_lists[order].free_list);
	list_add(&(merged_chunk -> node), free_list);
	/* BLANK END */

	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。

```c
static void choose_new_current_slab(struct slab_pointer *pool, int order)
{

	/* BLANK BEGIN */
	if (list_empty(&pool -> partial_slab_list)){
	  pool -> current_slab = init_slab_cache(order, SIZE_OF_ONE_SLAB); // no available slab, allocate new one
	} else {
	pool -> current_slab = list_entry(pool -> partial_slab_list.next, struct slab_header, node);  /* choose the
head->next in the partial_slab_list */
	list_del(pool -> partial_slab_list.next);
	}

	/* BLANK END */
}

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

	lock(&slabs_locks[order]);

	current_slab = slab_pool[order].current_slab;
	/* When serving the first allocation request. */
	/* unlikely is used to optimize branch, which tells compiler this case is unlikely to happen */
	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;
	}

	/* BLANK BEGIN */
	if (current_slab->current_free_cnt == 0) {			// no available slot!
		choose_new_current_slab(&slab_pool[order], order);
	}
	free_list = current_slab->free_list_head;
	current_slab->free_list_head = ((struct slab_slot_list *)(current_slab->free_list_head))->next_free;
	// free_list->next_free = NULL;
	--current_slab->current_free_cnt; // use one slot

	/* BLANK END */
```

```c
	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

	/* BLANK BEGIN */
	slot -> next_free = slab -> free_list_head; // do link list operation to free the slot
	slab -> free_list_head = (void *)slot;
	++ slab -> current_free_cnt;
	/* BLANK END */

	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

```c
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) {
		/* 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);
```

```c
                /* BLANK END */
        }

        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。

```c
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
{
        /* BLANK BEGIN */
  ptp_t *cur_ptp;
  pte_t *pte;
  int res;

  cur_ptp = (ptp_t *) pgtbl;
  /* L0 */
  res = get_next_ptp(cur_ptp, 0, va, &cur_ptp, &pte, false, NULL);
  if (res == -ENOMAPPING) {
    return res;
  }

  res = get_next_ptp(cur_ptp, 1, va, &cur_ptp, &pte, false, NULL);
  if (res == -ENOMAPPING) {
    return res;
  } else if (res == BLOCK_PTP) {
    if (entry != NULL){
      *entry = pte;
    }

    *pa = virt_to_phys(cur_ptp) + GET_VA_OFFSET_L1(va);
    return 0;
  }

  /* L2 */
  res = get_next_ptp(cur_ptp, 2, va, &cur_ptp, &pte, false, NULL);
  if (res == -ENOMAPPING) {
    return res;
  } else if (res == BLOCK_PTP) {

    if (entry != NULL){
      *entry = pte;
    }

    *pa = virt_to_phys(cur_ptp) + GET_VA_OFFSET_L2(va);
    return 0;
  }

  /* L3 */
  res = get_next_ptp(cur_ptp, 3, va, &cur_ptp, &pte, false, NULL);
  if (res == -ENOMAPPING) {
    return res;
  }
  if (entry != NULL){
    *entry = pte;
  }

  *pa = virt_to_phys(cur_ptp) + GET_VA_OFFSET_L3(va);
  return 0;


        /* BLANK END */

        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, long *rss)
{
        /* BLANK BEGIN */
  u64 page_num = len / PAGE_SIZE + (len % PAGE_SIZE > 0); /* ROUND_UP */
```

```
  while(page_num > 0){
    ptp_t *cur_ptp = (ptp_t *)pgtbl;
    pte_t *pte;
    for(int i = 0; i < 3; ++ i){
      get_next_ptp(cur_ptp, i, va, &cur_ptp, &pte, true, NULL);
    }

    for(int i = GET_L3_INDEX(va); i < PTP_ENTRIES; ++ i){
      pte_t new_pte;
      new_pte.pte = 0;
      new_pte.l3_page.is_valid = 1;
      new_pte.l3_page.is_page = 1;
      new_pte.l3_page.pfn = pa >> PAGE_SHIFT; /* page frame number */
      set_pte_flags(&new_pte, flags, kind);
      cur_ptp -> ent[i] = new_pte;


      -- page_num;

      if(page_num == 0){
        break;
      }

      va += PAGE_SIZE; /* move on to next mapping */
      pa += PAGE_SIZE;
    }

  }
        /* BLANK END */
        return 0;
}

int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, long *rss)
{
  /* BLANK BEGIN */
  u64 page_num = len / PAGE_SIZE + (len % PAGE_SIZE > 0); /* ROUND_UP */
  while (page_num > 0) {
        ptp_t *cur_ptp = (ptp_t *)pgtbl;
        pte_t *pte;
        int res;
       // L0
        res = get_next_ptp(cur_ptp, 0, va, &cur_ptp, &pte, false, NULL);
        if (res == -ENOMAPPING) {
                page_num -= L0_PER_ENTRY_PAGES;
                va += L0_PER_ENTRY_PAGES * PAGE_SIZE;
                continue;
        }
       // L1
        res = get_next_ptp(cur_ptp, 1, va, &cur_ptp, &pte, false, NULL);
        if (res == -ENOMAPPING) {
                page_num -= L1_PER_ENTRY_PAGES;
                va += L1_PER_ENTRY_PAGES * PAGE_SIZE;
                continue;
        }

        // L2
        res = get_next_ptp(cur_ptp, 2, va, &cur_ptp, &pte, false, NULL);
        if (res == -ENOMAPPING) {
                page_num -= L2_PER_ENTRY_PAGES;
                va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
                continue;
        }

        // L3
        for (int i = GET_L3_INDEX(va); i < PTP_ENTRIES; ++i) {
                cur_ptp->ent[i].pte = PTE_DESCRIPTOR_INVALID;

                va += PAGE_SIZE;

                --page_num;
                if (page_num == 0) {
                        break;
                }
            }
        }
```

```c
        /* BLANK END */

        dsb(ishst);
        isb();

        return 0;
}


int mprotect_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, vmr_prop_t flags)
{
   /* BLANK BEGIN */
  u64 page_num = len / PAGE_SIZE + (len % PAGE_SIZE > 0); /* ROUND_UP */
  while(page_num > 0){
    ptp_t *cur_ptp = (ptp_t *)pgtbl;
    pte_t *pte;
    for(int i = 0; i < 3; ++ i){
      get_next_ptp(cur_ptp, i, va, &cur_ptp, &pte, true, NULL);
    }

    for(int i = GET_L3_INDEX(va); i < PTP_ENTRIES; ++ i){

      set_pte_flags(&(cur_ptp -> ent[i]), flags, USER_PTE); // Set flags

      va += PAGE_SIZE; /* move on to next mapping */

      -- page_num;

      if(page_num == 0){
        break;
      }
    }

  }
        /* BLANK END */
        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）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 `01`（即可读可写），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。

粗粒度页表一次映射2M的页，可能会在后续使用过程中产生很多内部碎片。

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。

在填空部分填入以下一行内容即可。

```c
    ret = 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。

```c
struct vmregion *find_vmr_for_va(struct vmspace *vmspace, vaddr_t addr)
{
   /* BLANK BEGIN */
   // search the vmr which includes this va
  struct rb_node *node = rb_search(&(vmspace -> vmr_tree), (const void*)addr, cmp_vmr_and_va);
  if (node == NULL) {
    return NULL; // If not found
  }
  struct vmregion *res = rb_entry(node, struct vmregion, tree_node); // get the vmr that has node as its
`tree_node`
  return res;
  /* BLANK END */
}
```

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)
                /* kernel fault fixup is only supported on AArch64 and Sparc */
                return -EFAULT;
#endif
                sys_exit_group(-1);

                BUG("should not reach here");
        }

        pmo = vmr->pmo;
        /* Get the offset in the pmo for faulting addr */
        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: {
                /* Boundary check */
                BUG_ON(offset >= pmo->size);

                /* Get the index in the pmo radix for faulting addr */
                index = offset / PAGE_SIZE;

                fault_addr = ROUND_DOWN(fault_addr, PAGE_SIZE);

                pa = get_page_from_pmo(pmo, index);
                if (pa == 0) {

                        long rss = 0;

                        /* BLANK BEGIN */

                       pa = virt_to_phys((vaddr_t)get_pages(0)); // allocate a physical page
                       memset((void*)phys_to_virt(pa), 0, PAGE_SIZE); // clear it to 0
                        /* BLANK END */

                        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, &rss);
                /* BLANK END */
                vmspace->rss += rss;
                unlock(&vmspace->pgtbl_lock);
        } else {
                if (pmo->type == PMO_SHM || pmo->type == PMO_ANONYM) {
                        /* Add mapping in the page table */
                        long rss = 0;
                        lock(&vmspace->pgtbl_lock);
                        /* BLANK BEGIN */
                    map_range_in_pgtbl(vmspace -> pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss);
                        /* BLANK END */
                        /* LAB 2 TODO 7 END */
                        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;
}
```