ChCore 实验 2

班级：F2103003 姓名：黄伊新 学号：521030910070

1. **完成 buddy.c 中的 split_chunk、merge_chunk、buddy_get_pages、和 buddy_free_pages 函数**

首先是 split_chunk 函数，此处使用递归方法来进行实现。base case 是当传入的块的阶和 order 相同时，则直接返回该块，完成分裂。若不等，则需要将该块一分为二，并将其中的一块加入到 free list 中。在将该 chunk 的阶减 1 后即可通过 get_buddy_chunk 得到分裂后的后面一半的伙伴块，将其加入 free list 并进行有关设置。

```
static struct page *split_chunk(struct phys_mem_pool *pool, int
    order, struct page *chunk)
{
  struct list_head *free_list;
    struct page *buddy_page;
  if (chunk->order == order){ // if this is the wanted order
    return chunk;
  }
  chunk->order -= 1;

  free_list = &(pool->free_lists[chunk->order].free_list);
  buddy_page = get_buddy_chunk(pool, chunk); // split and get the
   remainder
  buddy_page->order = chunk->order;
  buddy_page->allocated = 0;
  list_append(&(buddy_page->node), free_list); // add the
   buddy_page to the free list
  pool->free_lists[chunk->order].nr_free += 1; // add one
  return split_chunk(pool, order, chunk); // recursively split the
     chunk
}

```

第二个实现的时 merge_chunk 函数。首先获得其对应的 buddy chunk，若传入 chunk 的阶没有超过最大限制，且能够找到一个空闲的伙伴块且阶数相同，则进行合并，若否则直接进行返回。需要注意的是在 merge 结束后是需要以编号更小的块作为新块的标识，这样才能通过得到块的位置来正确使用这个块的空间。

```
static struct page *merge_chunk(struct phys_mem_pool *pool, struct
     page *chunk)
{
  struct page *buddy;
```

```
  buddy = get_buddy_chunk(pool, chunk);
  if (chunk->order == BUDDY_MAX_ORDER - 1 || buddy == NULL ||
   buddy->allocated == 1 || buddy->order != chunk->order){
    return chunk;
  }
  list_del(&(buddy->node));
  pool->free_lists[buddy->order].nr_free -= 1;
  if ((u64) chunk < (u64) buddy){
    buddy->allocated = 1;
    chunk->order += 1; // already merge page with chunk
    return merge_chunk(pool, chunk);
  } else {
    chunk->allocated = 1;
    buddy->order += 1;
    return merge_chunk(pool, buddy);
  }
}

```

接下来完成的是 buddy_get_page 函数的实现。此处首先从 free list 中找到一个 best fit 的伙伴块，并进行分裂直到获得一个需要的阶数的伙伴块，并将该块标为已分配。

```
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(pool->free_lists[cur_order].free_list.next,
   struct page, node);
    list_del(&(page->node));
    pool->free_lists[cur_order].nr_free -= 1;
    break;
  }
}
if (page == NULL){
  unlock(&pool->buddy_lock);
  return NULL;
}

page = split_chunk(pool, order, page); // do the split
page->allocated = 1; // mark the page as allocated
page->order = order;

```

最后实现的是 buddy_free_page 的有关部分。这里主要需要处理一些特殊情况。若是正常的物理页且未分配则直接进行 merge，并将 merge 后的块加入 free list 并计数即可。

```
if (page == NULL) {return;}
if (page -> allocated == 0) {return;}
page->allocated = 0; // marked as free
```

```
page = merge_chunk(pool, page); // merge chunk and return the
    merged one
order = page->order; // the order of the merged chunk
free_list = &(pool->free_lists[order].free_list);
list_append(&(page->node), free_list); // insert the free page
    into the free list
pool->free_lists[order].nr_free += 1; // do the updates

```

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**

第一个实现的是 choose_new_current_slab 函数。主要逻辑是若 partial list 中没有有空余的 slab 则从 buddy system 中直接获取一个新的，若有则取出并进行分配。从 partial list 中转换成 slab 使用的是 list_entry 这个宏。

```
static void choose_new_current_slab(struct slab_pointer *pool, int
    order)
{
  if (list_empty(&pool->partial_slab_list)) { // all slabs full
    pool->current_slab = init_slab_cache(order, SIZE_OF_ONE_SLAB);
    // get one from buddy system
  } else {
    pool->current_slab = list_entry(pool->partial_slab_list.next,
    struct slab_header, node); // change current slab to the first
    in the partial slab list
    list_del(&pool->current_slab->node); // remove it from the
    list
  }
}

```

第二个是实现 alloc_in_slab_impl 函数中的内容。此处首先判断目前使用的 slab 中是否有空闲，若否则从 partial list 中选择一个作为新的 slab。接着，从 current slab 中取出一个空闲的 slot，并进行有关更新。

```
if (current_slab->free_list_head == NULL){ // no free slot in
    current slab
  choose_new_current_slab(&slab_pool[order], order); // choose a
    new slab
  current_slab = slab_pool[order].current_slab; // change to the
    new slab
}
free_list = (struct slab_slot_list *)current_slab->free_list_head;
    // the first free slot in the current slab
next_slot = free_list->next_free; // the second free slot/NULL
current_slab->free_list_head = next_slot; // remove the slot from
    the slab
current_slab->current_free_cnt -= 1; // update the counts

```

最后是 free_in_slab 函数的有关实现。本处直接将 slot 插入回 free list，并计数即可。

```
slot->next_free = slab->free_list_head;
slab->free_list_head = (void *)slot; // insert slot to the
    free_list
slab->current_free_cnt += 1; // update the number of free slots

```

3. **练习题 3：完成 __kmalloc 函数，在适当位置调用对应函数，实现有关功能**

本处主要是进行函数调用，若 size 较小则调用 slab 来进行分配，若较大则首先通过函数来计算出 best fit 的阶，然后使用 buddy system 进行分配。

```
if (size <= SLAB_MAX_SIZE) {
  addr = alloc_in_slab(size, real_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);
}

```

4. **练习题 4：完成 query_in_pgtbl, map_range_in_pgtbl_common, 和 unmap_range_in_pgtbl 和 mprotext_in_pgtbl 函数**

首先是 query_in_pgtbl 函数。实现的思路大致是不断进行 get_next_ptp 函数的调用，若返回为-ENOMAPPING，即未映射，则直接进行返回；若碰到的是块描述符，则返回块描述符的内容和地址；若否则接着进行下一级页表的访问。本初因为代码较长就不放入 report 中了。

接着是 map_range_in_pgtbl_common 函数。该函数完成的任务是将一个区间内的虚拟地址空间映射到一个区间内的物理地址上。首先需要计算需要几个物理页来承载这些地址。接着依次来分配这些物理页。具体来说是通过 get_next_ptp 来获得/分配页表页，并具体来填写每一个页表项，并对已填写的页表项进行计数，若已达目标则退出。

然后是 unmap_range_in_pgtbl 函数的填写。本处的进行方式同上面函数类似，都是先计算需要删除映射的数目，并依次进行。需要特别注意的是本处可能遇到本身就有没映射的情况，那么在每一层获取 page 的时候都需要额外进行判断，若该层本来就没有进行映射那么就不用再删除其下的项。若是最后发现有这个映射，那只需要在 l3 页表中将 valid 位设 0 就行。

最后是 mprotect_in_pgtbl 函数。这里主要是对最后一层的权限进行修改。方式和前面类似，本处直接到最后一层，并进行权限的修改即 set_pte_flags。

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需要配置页表描述符的哪个字段，并在发生页错误时怎么处理**

   在操作系统中支持写时拷贝需要配置页表描述符的 AP 字段，在 manual 的 B4.3 这一板块中。AP 即 Access Permission，它指定了访问该页的权限。在写时拷贝中，AP 字段应该设置为只读，以便多个进程可以共享同一块内存。

   当进程试图写入只读页时，会触发一个页错误。当操作系统发现本处是由于本处是由于写时拷贝而触发的异常 Permission Fault 时，其会分配一个新的物理页，并将该页的内容复制到新的物理页中。然后，操作系统会更新页表，以便该页指向新的物理页。这样，该进程就可以修改其私有副本，而不会影响其他进程。

6. **思考题 6：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？**

   这导致了后续只能分配一个较大的页，较大的页可能会产生较多的内部碎片，内存的利用率较低。

7. **挑战题 7：使用前面实现的函数，配置细粒度页表**

   挑战题本处没有在代码中加入这一部分，仅在报告中呈现一下大概的方式。

```
void reconfig_kernel_page_table() {
    // memory
    vmr_prop_t flags = AARCH64_MMU_ATTR_PAGE_AP_HIGH_RW_EL0_RW |
   AARCH64_MMU_ATTR_PAGE_UX | NORMAL_MEMORY;
    void *pgtbl;
    pgtbl = get_pages(0);
    memset(pgtbl, 0, PAGE_SIZE);
    int ret;
    size_t len1 = (1 << 30) - (1 << 24);
    map_range_in_pgtbl(pgtbl, KBASE, 0, len1, flags);

    // device memory
    flags = AARCH64_MMU_ATTR_PAGE_AP_HIGH_RW_EL0_RW |
   AARCH64_MMU_ATTR_PAGE_UX | DEVICE_MEMORY;
    size_t len2 = 1 << 24;
    map_range_in_pgtbl(pgtbl, KBASE + len1, len1, len2, flags);

    // unmap
    set_page_table(pgtbl);
}

```

8. **练习题 8：完成 do_page_fault 函数，将缺页异常进行转发**

   这里直接调用 handle_trans_fault 函数即可，传入的函数就是本进程对应的 vmspace 和发生错误的地址。

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

```

9. **练习题 9：完成 find_vmr_for_va 函数，找到一个虚拟地址在其虚拟地址空间中的 VMR**

这里是通过红黑树的搜索函数来获得包含该虚拟地址的 vmregion，调用的函数为 rb_search，使用的是函数 cmp_vmr_and_va 来进行比较。最后通过 rb_entry 函数来获得包含该节点的 vmregion 即为我们所找的 vmregion。

```
struct vmregion *find_vmr_for_va(struct vmspace *vmspace, vaddr_t
    addr)
{
  struct rb_root *vmr_root = &vmspace->vmr_tree;
  struct rb_node *rb;
  rb = rb_search(vmr_root, (const void*)addr, cmp_vmr_and_va);
  if (rb == NULL) {
      return NULL;
    }
    return rb_entry(rb, struct vmregion, tree_node);
}

```

10. **练习题 10：完成 handle_trans_fault 函数，实现按需物理页分配**

若 pa==0，则可以判断这一部分是需要进行按需分配的，则首先通过 get_pages 来获得一个新的物理页及其物理地址，接着再将这一部分的物理页加入到页表中即可。

若否，那么这里是被移出的页，根据提示我们需要做的就是就是将这一部分页表重新进行映射，调用 map_range_in_pgtbl 即可。

```
if (pa == 0) {
    long rss = 0;
    /* LAB 2 TODO 7 BEGIN */
    /* BLANK BEGIN */
    /* Hint: Allocate a physical page and clear it to 0. */
    pa = virt_to_phys((vaddr_t)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, 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);
    }
}

```

11. **挑战题 11：简单实现换页技巧**

本处选择最简单的方式实现换页。假设作为内存的物理页从 0 开始，而模拟硬盘的在高地址。本处认为共有 1G 作为内存的地址，页大小为 4K，故总共有 2^{18} 的页数。

接着使用数组来记录，即课本中提到的用一个位图来记录物理页是否有被映射，在换出时需要额外写一个函数来分配在指定区域内的一个物理页。具体实现如下。

```
#define page_num_in_memory (1 << 18)// number of pages in physical
    memory

bool is_mapped[page_num_in_memory];
vaddr_t origin_va[page_num_in_memory];
vmr_prop_t origin_perm[page_num_in_memory];
u64 access_time[page_num_in_memory]; // record the last time that
   it got accessed
u64 current_time = 0;


void alloc_in_memory(void *pgtbl, vaddr_t va, paddr_t pa,
   vmr_prop_t perm, long *rss) {
  // a simple implementation of LRU
        u64 min_time = -1;
        u64 alloc_index;

        for (u64 i = 0; i < page_num_in_memory; ++i) {
                if (is_mapped[i]) {
                        if (min_time > access_time[i]) {
                                min_time = access_time[i];
```

```
                                alloc_index = i;
                        }
                }
                else {
                  // empty page available
                        alloc_index = i;
                        break;
                }
        }

        if (is_mapped[alloc_index]) {
                // swap out
                paddr_t pa_swap = get_pages_in_disk(); // to be
realized
                unmap_range_in_pgtbl(pgtbl, origin_va[alloc_index
], PAGE_SIZE, rss);
                memcpy(phys_to_virt(pa_swap), (void *)phys_to_virt
(alloc_index * PAGE_SIZE), PAGE_SIZE);
                map_range_in_pgtbl(pgtbl, origin_va[alloc_index],
pa_swap, PAGE_SIZE, origin_perm[alloc_index], rss);
        }

        // allocate
        is_mapped[alloc_index] = true;
        origin_va[alloc_index] = va;
        origin_perm[alloc_index] = perm;
        access_time[alloc_index] = current_time++;

        memcpy(phys_to_virt(alloc_index * PAGE_SIZE), (void *)
phys_to_virt(pa), PAGE_SIZE);
        map_range_in_pgtbl(pgtbl, va, alloc_index * PAGE_SIZE,
PAGE_SIZE, perm, rss);
}

```