实验二： 内存管理

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`
  - 代码补全：

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


        struct  page *buddy_chunk;

        chunk -> order = chunk -> order - 1;


        buddy_chunk = get_buddy_chunk(pool,chunk);
        buddy_chunk->order = chunk -> order ;
        list_add(& buddy_chunk->node,& pool->free_lists[buddy_chunk->order].free_list);
        pool->free_lists[buddy_chunk->order].nr_free ++;
        buddy_chunk -> pool = chunk -> pool;
        buddy_chunk -> allocated = 0;


        BUG_ON(chunk == NULL);
        return split_chunk(pool, order, chunk);

```

  - **思路：**

    根据提示使用递归，如果当前chunk的order为所需order直接返回，

    如果不是，则需要进一步split：

- 先将order - 1， 获取对应的buddy_chunk
- 对得到的buddy_chunk,对属性进行赋值，并加入到对应的free_list中
- 最后递归调用 split

注意：当把buddy_chunk加入到对应的free_list中需要对nr_free++

`merge_chunk`

核心代码补全：

```
        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 the buddy chunk is not free, no further merge is requiied

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

        //  buddy_chunk is not free as a whole
        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);
```

思路：（这一部分主要参考了tutorial视频）根据提示，进行递归实现

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

在递归调用merge

`buddy_get_page`

- 核心代码：

```
        cur_order = order;

        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;


                        page = list_entry(free_list, struct page, node);
                        pool->free_lists[page->order].nr_free --;

                        page = split_chunk(pool,order,page);
                        list_del(&(page->node));
                        page ->allocated = 1;
                        break;
                }
        }
```

- **思路：**

  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`

- 代码补全：(主要参考tutorial视频)

```
        page -> allocated = 0;
        //  Merge the freed chunk
        page = merge_chunk(pool, page);

        // Put the merged chunk into the 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;
```

- 思路：

  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

(为节省篇幅，之后只对代码思路以及需要注意的地方进行阐述，不再对补全核心代码进行引用)

`choose_new_current_slab`

- **思路：**

  利用 `pool->partial_slab_list` 获得对应的partial slab链

  - 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`

- **思路：**

  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`

- **思路：** 将已分配的slot 释放，只要将alloct的过程逆向即可

练习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

- 思路：
  - 分别调用已有的 `alloc_in_slab` 分配小块的内存
  - 调用 `get_pages` 分配大块内存即可

注意：需要利用`size_to_page_order`将size转化为对应大小的order

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。

- 思路：

  `query_in_pgtbl`：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

  注意：1) 区分物理地址和页表项entry 2) get_next_ptp 应当alloct设置为false

  `map_range_in_pgtbl_common`

  - 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`
  - 作为`map_range_in_pgtbl_common`的逆过程
  - 关键在将get_next_ptp 应当alloct设置为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

    tip:unmap操作通过对pte 页表项赋值为 0 实现

- `mprotect_in_pgtbl`
  - 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：

> 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）[^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）

解答：

- 为写时拷贝支持需要配置：
  - **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 （查阅资料得到）
- 触发页错误时，确定是有写时拷贝机制触发的异常
  - 索引到对应的物理页，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

```

        int ret = 0;
        paddr_t pa;
        pte_t *fault_pte;
        struct common_pte_t pte_info;
        lock(&vmspace->pgtbl_lock);
        ret = query_in_pgtbl(vmspace->pgtbl, fault_addr, &pa,
&fault_pte);
        /**
         * Although we are handling a permission fault here, it's still
possible
         * that we would discover it should be a translation error when
we
         * really started to resolve it here. For example, before the
page fault
         * was forwarded to here, we didn't grab the pgtbl_lock, so it's
         * possible that the page was unmapped or swapped out. There is
another
         * special case: in RISC-V, we are unable to determine a page
fault is
         * translation fault or permission fault from scause reported by
the
         * hardware, so we have to check the pte here.
         *
         * We query the page table for the actual PTE **atomically**
here. So if
         * the PTE is missing, we can be sure that it's a translation
fault. And
         * we forward it back to translation fault handler by returning
-EFAULT.
         */
        if (ret) {
                ret = -EFAULT;
                goto out;
        }
        parse_pte_to_common(fault_pte, L3, &pte_info);

        // Fast path: already handled page fault in other threads
        if (pte_info.perm & VMR_WRITE) {
                goto out;
        }
        ret = __do_general_cow(
                vmspace, fault_vmr, fault_addr, fault_pte, &pte_info);
```

```
34
35  out:
36          unlock(&vmspace->pgtbl_lock);
37          return ret;
```

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

> 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。

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

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*。

思路：在理解vmregion、vmspace的基础上，调用红黑树的宏函数 `rb_search`，`rb_entry` 即可找到对应的vmr

```
1  struct rb_node *node =  rb_search(&(vmspace->vmr_tree),addr,cmp_vmr_and_va);
2          if(node == NULL) return node;
3          return rb_entry(node,struct vmregion, tree_node);
```

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。

思路：

- 根据提示
  - 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
- 关键在于区分不同的处理页表异常的情况

附录： 涉及的数据结构整理：

Buddy System：

list_head： 双向链表：

- 初始化

```
static inline void init_list_head(struct list_head *list)
```

- 增加元素到头部

```
static inline void list_add(struct list_head *new, struct list_head *head)
```

- 增加元素到尾

```
static inline void list_append(struct list_head *new, struct list_head *head)
```

-
  - 删除节点：

```
static inline void list_del(struct list_head *node)
```

  - 是否为空：

```
static inline bool list_empty(struct list_head *head)
```

  - list_entry： **获得节点对应的元素**

```
list_entry(ptr, type, field)
```

    ptr: 指向节点的指针

    type: 元素的数据类型

    field: 节点所在的field

  - for_each_in_list: 遍历列表中的元素

```
#define for_each_in_list(elem, type, field, head) \
    for ((elem) = container_of((head)->next, type, field); \
         &((elem)->field) != (head); \
         (elem) = container_of(((elem)->field).next, type, field))
```

    elem： 存放元素

    type: 元素的数据类型

    field: 节点所在的field

head: 链表头部的指针

page:

```
/* struct page is the metadata of one physical 4K page. */
struct page {
        /* Free list */
        struct list_head node;
        /* Whether the correspond physical page is free now. */
        int allocated;
        /* The order of the memory chunck that this page belongs to. */
        int order;
        /* Used for ChCore slab allocator. */
        void *slab;
        /* The physical memory pool this page belongs to */
        struct phys_mem_pool *pool;
};
```

free_list: 用于phys_mem_pool中的放相同order的chunk

```
struct free_list {
        struct list_head free_list;
        unsigned long nr_free;
};
```

phys_mem_pool:

```
struct phys_mem_pool {
        /*
         * The start virtual address (for used in kernel) of
         * this physical memory pool.
         */
        vaddr_t pool_start_addr;
        unsigned long pool_mem_size;

        /*
         * The start virtual address (for used in kernel) of
         * the metadata area of this pool.
         */
        struct page *page_metadata;

        /* One lock for one pool. */
        struct lock buddy_lock;

        /* The free list of different free-memory-chunk orders. */
        struct free_list free_lists[BUDDY_MAX_ORDER];

        /*
         * This field is only used in ChCore unit test.
         * The number of (4k) physical pages in this physical memory pool.
         */
        unsigned long pool_phys_page_num;
};
```

- 把pool看成整个伙伴系统的资源池
- free_lists 是存放各个order的chunk的数组

Slab 分配器：

slab_header:

```
/* slab_header resides in the beginning of each slab (i.e., occupies the first slot). */
struct slab_header {
        /* The list of free slots, which can be converted to struct slab_slot_list. */
        void *free_list_head;
        /* Partial slab list. */
        struct list_head node;

        int order;
        unsigned short total_free_cnt; /* MAX: 65536 */
        unsigned short current_free_cnt;
};
```

- free_list_head： 看成是next_free

slab_slot_list

```
39
40  /* Each free slot in one slab is regarded as slab_slot_list. */
41  struct slab_slot_list {
42          void *next_free;
43  };
```

slab_pointer

- ```
  44
  45  struct slab_pointer {
  46          struct slab_header *current_slab;
  47          struct list_head partial_slab_list;
  48  };
  ```
  - 一个确定大小slot 的slab系统

slab_pool

```
struct slab_pointer slab_pool[SLAB_MAX_ORDER + 1];
```

- 管理不同slot大小的slab系统的资源池

Slab内存分配辅助函数：

- page 和 virture addr 互转：

```
void *page_to_virt(struct page *page)
```

```
9   struct page *virt_to_page(void *ptr)
```
```

- order 和 size 互转：

```
static inline int size_to_order(unsigned long size)
static inline unsigned long order_to_size(int order)
```

- set_or_clear_slab_in_page

  把addr开始的size 大小的pages的slab都置为NULL，或者addr

- alloc_slab_memory:

```
81
82  static void *alloc_slab_memory(unsigned long size)
83  {
```

页表管理：

-

```
ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false, NULL);
if(ret == -ENOMAPPING)
```

  - 从当前的页表页，获取对应va的下一级页表页 -> next_ptp
  - pte是对应的页表项，还有valid位、物理页号等
  - allocat：判断如果遇到未分配的页表页，是否进行分配

vmregion

```
21
22  /* This struct represents one virtual memory region inside on address space */
23  struct vmregion {
24          struct list_head list_node; /* As one node of the vmr_list */
25          struct rb_node tree_node; /* As one node of the vmr_tree */
26          /* As one node of the pmo's mapping_list */
27          struct list_head mapping_list_node;
28
29          struct vmspace *vmspace;
30          vaddr_t start;
31          size_t size;
32          /* Offset of underlying pmo */
33          size_t offset;
34          vmr_prop_t perm;
35          struct pmobject *pmo;
36          struct list_head cow_private_pages;
37  };
38
```

- vmspace：所属的vmspace
- start：对应虚拟区域的起始地址
- size: 对应的虚拟区域的大小
- prem: vmr permissions

vmspace

```c
/* This struct represents one virtual address space */
struct vmspace {
        /* List head of vmregion (vmr_list) */
        struct list_head vmr_list;
        /* rbtree root node of vmregion (vmr_tree) */
        struct rb_root vmr_tree;

        /* Root page table */
        void *pgtbl;
        /* Address space ID for avoiding TLB conflicts */
        unsigned long pcid;

        /* The lock for manipulating vmregions */
        struct lock vmspace_lock;
        /* The lock for manipulating the page table */
        struct lock pgtbl_lock;

        /*
         * For TLB flushing:
         * Record the all the CPU that a vmspace ran on.
         */
        unsigned char history_cpus[PLAT_CPU_NUM];

        struct vmregion *heap_boundary_vmr;

        /* Records size of memory mapped. Protected by pgtbl_lock. */
        unsigned long rss;
};
```