ChCore Lab2 课程作业

2023 年 11 月 13 日

题目 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。

```
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;
        }
```

```
        chunk->order -= 1;
        struct page *buddy_chunk;
        buddy_chunk = get_buddy_chunk(pool, chunk);

        buddy_chunk->order = chunk->order;
        buddy_chunk->allocated = 0;
        list_add(&(buddy_chunk->node),
        &(pool->free_lists[buddy_chunk->order].free_list));
        pool->free_lists[buddy_chunk->order].nr_free += 1;

        return split_chunk(pool, order, chunk);

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

题目 1 的注记. split_chunk 可以通过分裂获取指定阶数的伙伴块, 以递归的方式实现。首先判断基本情况：目前的 chunk 正好满足所需阶数。否则分裂当前 chunk，先将当前 chunk 的阶数减一，然后获取其伙伴块，这样得到的两个新块都是原 chunk 阶数减一，这也就实现了分裂原 chunk。再将伙伴块加入空闲链表，将 chunk 传入 split_chunk 实现递归调用。

```
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 */
struct page *buddy_chunk;

if (chunk->order == (BUDDY_MAX_ORDER - 1)){
        return chunk;
}

buddy_chunk = get_buddy_chunk(pool, chunk);

if (buddy_chunk == NULL)
        return chunk;

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

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);

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

题目 1 的注记. merge_chunk 可以通过合并获得指定阶数的伙伴块，首先判定一些不能合并的基本情况；如果可以合并，获取其伙伴块并将二者阶数加一，最终将二者中的低地址作为大块的地址，递归调用 merge_chunk 函数

```
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);
```

```
        /* LAB 2 TODO 1 BEGIN */
        /*
         * Hint: Find a chunk that satisfies the order requirement
         * in the free lists, then split it if necessary.
         */
        /* 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;
        page = list_entry(free_list, struct page, node);
        pool -> free_lists[page -> order].nr_free -= 1;
        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 */
        /* LAB 2 TODO 1 END */
out:
        unlock(&pool->buddy_lock);
        return page;
```

}

题目 1 的注记. buddy_get_pages 从物理内存池中得到指定阶数的块，按阶数从小到大遍历空闲链表，直至找到空闲块，再利用先前的 split_chunk 函数对空闲块进行拆分。

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

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

题目 1 的注记. buddy_free_pages 用于释放内存块，先将刚释放的物理页合并，然后加入对应阶数的空闲链表。

题目 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。

```
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 */
}
```

题目 2 的注记. choose_new_current_slab 用于选择新的当前 slab，从 partial slab 链表中获取头节点并获取到对应的 slab header，并将其从收 partial 链表中删除，令 current 指针指向它。

```
static void *alloc_in_slab_impl(int order)
{
...
        /* LAB 2 TODO 2 BEGIN */
        /* 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 -= 1;

        if (unlikely(current_slab->current_free_cnt == 0))
                choose_new_current_slab(&slab_pool[order]);

        /* BLANK END */
```

```
        /* LAB 2 TODO 2 END */

        unlock(&slabs_locks[order]);

        return (void *)free_list;
}
```

题目 2 的注记. alloc_in_slab_impl 用于分配指定阶数的 slab，只需调用 choose_new_current_slab 函数即可实现。

```
void free_in_slab(void *addr)
{
...
        /* LAB 2 TODO 2 BEGIN */
        /* BLANK BEGIN */
        slot -> next_free = slab -> free_list_head;
        slab -> free_list_head = (void *)slot;
        slab -> current_free_cnt += 1;
        /* BLANK END */
        /* LAB 2 TODO 2 END */

        try_return_slab_to_buddy(slab, order);

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

题目 2 的注记. free_in_slab，释放 slot，将 slot 加在空闲 slot 表的首位，并将空闲 slot 数加一。

题目 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

```
/* Currently, BUG_ON no available memory. */
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);
                /* BLANK END */
                /* LAB 2 TODO 3 END */
        }

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

题目 3 的注记. 分两种情况考虑，小于 MAX_SLAB 大小用 slab 分配器分配，大于 MAX_SLAB 用伙伴系统分配实现。

题目 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。

```
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa,
        pte_t **entry)
{
        /* LAB 2 TODO 4 BEGIN */
        /* BLANK BEGIN */

        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
        pte_t *pte, *pe;
```

```
int ret;
int pte_index;

BUG_ON(pgtbl == NULL);
//BUG_ON(va % PAGE_SIZE);


l0_ptp = (ptp_t *)pgtbl;

l1_ptp = NULL;
l2_ptp = NULL;
l3_ptp = NULL;
pe = NULL;
// l0
ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp,
        &pte, false, NULL);
BUG_ON(ret != 0);
if (ret == -ENOMAPPING) {
        return ret;
}

// l1
ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp,
        &pte, false, NULL);
BUG_ON(ret != 0);
if (ret == -ENOMAPPING) {
        return ret;
```

```
        }

        // l2
        ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp,
                &pte, false, NULL);
        BUG_ON(ret != 0);
        if (ret == -ENOMAPPING) {
                return ret;
        }

        // l3
        ret = get_next_ptp(l3_ptp, 3, va, &pe,
                &pte, false, NULL);
        if (ret == -ENOMAPPING) {
                return ret;
        }
        if (entry != NULL){
                *entry = pte;
        }

        *pa = virt_to_phys(pe) + GET_VA_OFFSET_L3(va);

        /* BLANK END */
        /* LAB 2 TODO 4 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)
{
        /* LAB 2 TODO 4 BEGIN */
        /* 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){
                // 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
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;
                }
        }

        /* BLANK END */
        /* LAB 2 TODO 4 END */
        return 0;
}

int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va,
        size_t len, long *rss)
{
        /* LAB 2 TODO 4 BEGIN */
        /* 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){
        // 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
        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;
                        if (rss)
                                *rss += PAGE_SIZE;
                        total_page_cnt -= 1;
                        if (total_page_cnt == 0)
                                break;
                }
        }

        /* BLANK END */
        /* LAB 2 TODO 4 END */

        dsb(ishst);
        isb();

        return 0;
}

int mprotect_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, vmr_prop_t
{
        /* LAB 2 TODO 4 BEGIN */
        /* 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){
        // l0
        ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp,
                &pte, true, NULL);
        BUG_ON(ret != 0);

        // l1
        ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp,
                &pte, true, NULL);
        BUG_ON(ret != 0);

        // l2
        ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp,
                &pte, true, NULL);
        BUG_ON(ret != 0);
```

```
                // l3
                pte_index = GET_L3_INDEX(va);
                for(i = pte_index; i < PTP_ENTRIES; ++i){
                        pte_t new_pte_val = l3_ptp->ent[i];
                        set_pte_flags(&new_pte_val, 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;
}
```

题目 4 的注记. 上述四个函数的基本思路都是一致的，通过走页表获取目标 entry，并对 entry 进行查询或更改即可。

题目 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）

解答. L3 页表项中的 AP 字段用于定义物理页的读写权限。

Copy-on-Write 物理页应设为可读不可写，即 AP 字段为 11 。当有应用程序尝试写物理页时，会引发一个访问权限异常。此时，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），并映射给触发异常的应用程序。

题目 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。

解答. 粗粒度页表一次映射 2M 的大页，可能导致资源浪严重，内部碎片多。

题目 7.

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。

题目 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。

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

题目 8 的注记. 查阅 handle_trans_fault 的函数头可知，应使用如上的参数。

题目 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。

```
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 vmregion* vmr;
        struct rb_node * rbn;


        //list_add(&vmr->list_node, &vmspace->vmr_list);
        rbn = rb_search(&vmspace->vmr_tree, (const void*)addr,
                cmp_vmr_and_va);
        if (rbn == NULL)
        {
                return NULL;
        }
        vmr = rb_entry(rbn, struct vmregion, tree_node);
        return vmr;
        /* BLANK END */
        /* LAB 2 TODO 6 END */
}
```

题目 9 的注记. 先利用 rb_search 函数实现对红黑树节点的搜索，通过查阅 rb_search 函数头得知，要想实现按地址搜索，应当使用 cmp_vmr_and_va 作为比较器；然后将红黑树节点转化为 vmregion 结构体。

题目 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。

```
int handle_trans_fault(struct vmspace *vmspace, vaddr_t fault_addr)
{
...
    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((void*)phys_to_virt(pa), 0, PAGE_SIZE);
        /* BLANK END */
        ...
        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);
            }
...
}
```

题目 10 的注记.

对于未分配物理内存的页，应当先利用 get_pages 实现分配，再利用 menset 实现置 0;

对于已分配物理内存的页，只需建立内存映射即可，我们可以利用之前实现的 map_range_in_pgtbl.

题目 11.

挑战题 11: 由于没有磁盘，因此我们采用一部分内存模拟磁盘。内存

页是可以换入换出的，请设计一套换页策略（如 LRU 等），并在 ‘kernel/mm/pgfault_handler.c‘ 中的的适当位置简单实现你的换页方法。