实验 2：内存管理

薛翔元 (521030910387)

> 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

对于 `split_chunk` 和 `merge_chunk` 两个函数，我们采用递归的方式实现，在 `chunk` 分裂或合并后，需要及时维护其 `allocated` 和 `order` 字段，并在相应链表中更新空闲块信息。

```
/* LAB 2 TODO 1 BEGIN */
/*
* Hint: Recursively put the buddy of current chunk into
* a suitable free list.
*/
if (chunk->order == order) {
        return chunk;
} else {
        --chunk->order;
        struct page *buddy = get_buddy_chunk(pool, chunk);
        buddy->allocated = 0;
        buddy->order = chunk->order;
        list_add(&buddy->node, &pool->free_lists[buddy->order].free_list);
        ++pool->free_lists[buddy->order].nr_free;
        return split_chunk(pool, order, chunk);
}
/* LAB 2 TODO 1 END */
```

```
/* LAB 2 TODO 1 BEGIN */
/*
* Hint: Recursively merge current chunk with its buddy
* if possible.
*/
if (chunk->order == BUDDY_MAX_ORDER - 1) {
        return chunk;
}

struct page *buddy = get_buddy_chunk(pool, chunk);
if (buddy == NULL || buddy->allocated == 1 || buddy->order != chunk->order) {
```

```
        return chunk;
} else {
        list_del(&buddy->node);
        --pool->free_lists[buddy->order].nr_free;
        if (chunk > buddy) {
                chunk = buddy;
        }
        ++chunk->order;
        return merge_chunk(pool, chunk);
}
/* LAB 2 TODO 1 END */
```

对于 `buddy_get_pages` 函数，我们首先遍历 `free_lists`，找到第一个足够大的空闲块，将其分裂至指定阶，作为分配的内存块。

```
/* LAB 2 TODO 1 BEGIN */
/*
    * Hint: Find a chunk that satisfies the order requirement
    * in the free lists, then split it if necessary.
    */
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;
                page = list_entry(free_list->next, struct page, node);
                break;
        }
}

if (page == NULL) {
        goto out;
}

list_del(&page->node);
--pool->free_lists[cur_order].nr_free;
page = split_chunk(pool, order, page);
page->allocated = 1;
/* LAB 2 TODO 1 END */
```

对于 `buddy_free_pages` 函数，我们只需要将待释放的内存块进行合并，同时维护相应信息即可。

```
/* LAB 2 TODO 1 BEGIN */
/*
    * Hint: Merge the chunk with its buddy and put it into
    * a suitable free list.
    */
page->allocated = 0;
page = merge_chunk(pool, page);
list_add(&page->node, &pool->free_lists[page->order].free_list);
++pool->free_lists[page->order].nr_free;
/* LAB 2 TODO 1 END */
```

> 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

对于 `choose_new_current_slab` 函数，我们从相应的 `partial_slab_list` 中取出队首，作为新的 `current_slab` 即可。值得注意的是，当 `partial_slab_list` 为空时，我们直接将新的 `current_slab` 设置为 `NULL`，空置造成的问题将由 `alloc_in_slab_impl` 函数解决。

```
/* LAB 2 TODO 2 BEGIN */
/* Hint: Choose a partial slab to be a new current slab. */
if (list_empty(&pool->partial_slab_list)) {
        pool->current_slab = NULL;
} else {
        pool->current_slab = list_entry(pool->partial_slab_list.next, struct
slab_header, node);
        list_del(pool->partial_slab_list.next);
}
/* LAB 2 TODO 2 END */
```

对于 `alloc_in_slab_impl` 函数，我们首先处理 `current_slab` 为 `NULL` 时的内存申请，然后从 `current_slab` 中取出第一个 `slot` 作为分配的内存，若此时 `current_slab` 已满则调用 `choose_new_current_slab` 函数重新进行选择。

```
/* LAB 2 TODO 2 BEGIN */
/*
    * Hint: Find a free slot from the free list of current slab.
    * If current slab is full, choose a new slab as the current one.
    */
free_list = (struct slab_slot_list *)(current_slab->free_list_head);
next_slot = free_list->next_free;
current_slab->free_list_head = next_slot;
if (--current_slab->current_free_cnt == 0) {
        choose_new_current_slab(&slab_pool[order], order);
}
/* LAB 2 TODO 2 END */
```

对于 `free_in_slab` 函数，我们将待释放的 `slot` 放回相应的 `slab` 即可，若此时该 `slab` 未满则将其插入 `partial_slab_list` 中，若此时该 `slab` 已空则将其返还给伙伴系统。

```
/* LAB 2 TODO 2 BEGIN */
/*
    * Hint: Free an allocated slot and put it back to the free list.
    */
slot->next_free = slab->free_list_head;
slab->free_list_head = (void *)slot;
++slab->current_free_cnt;
/* LAB 2 TODO 2 END */
```

> 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

当申请内存的大小超过临界值时，我们调用 `get_pages` 函数从伙伴系统中分配大块内存，否则调用 `alloc_in_slab` 分配小块内存。

```
if (size <= SLAB_MAX_SIZE) {
    /* LAB 2 TODO 3 BEGIN */
    /* Step 1: Allocate in slab for small requests. */
    addr = alloc_in_slab(size, real_size);
} else {
    /* Step 2: Allocate in buddy for large requests. */
    order = size_to_page_order(size);
    addr = _get_pages(order, is_record);
    /* LAB 2 TODO 3 END */
}
```

> 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。

对于 `query_in_pgtbl` 函数，我们反复调用 `get_next_ptp` 函数，直到找到相应的页表项。

```
/* LAB 2 TODO 4 BEGIN */
/*
    * Hint: Walk through each level of page table using `get_next_ptp`,
    * return the pa and pte until a L0/L1 block or page, return
    * `-ENOMAPPING` if the va is not mapped.
    */
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
ptp_t *phys_page;
pte_t *pte;
int ret;
```

```
l0_ptp = (ptp_t *)pgtbl;
ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false, NULL);
if (ret < 0) {
        return ret;
}

ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false, NULL);
if (ret < 0) {
        return ret;
} else if (ret == BLOCK_PTP) {
        *pa = (pte->l1_block.pfn << L1_INDEX_SHIFT) | (GET_VA_OFFSET_L1(va));
        if (entry != NULL) {
                *entry = pte;
        }
        return 0;
}

ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false, NULL);
if (ret < 0) {
        return ret;
} else if (ret == BLOCK_PTP) {
        *pa = (pte->l2_block.pfn << L2_INDEX_SHIFT) | (GET_VA_OFFSET_L2(va));
        if (entry != NULL) {
                *entry = pte;
        }
        return 0;
}

ret = get_next_ptp(l3_ptp, L3, va, &phys_page, &pte, false, NULL);
if (ret < 0) {
        return ret;
}
*pa = (pte->l3_page.pfn << L3_INDEX_SHIFT) | (GET_VA_OFFSET_L3(va));
if (entry != NULL) {
        *entry = pte;
}
/* LAB 2 TODO 4 END */
```

对于 `map_range_in_pgtbl_common`、`unmap_range_in_pgtbl`、`mprotect_in_pgtbl` 三个函数，我们遍历指定内存区间，依次配置相应的页表项即可。

```
/* LAB 2 TODO 4 BEGIN */
/*
    * Hint: Walk through each level of page table using `get_next_ptp`,
    * create new page table page if necessary, fill in the final level
    * pte with the help of `set_pte_flags`. Iterate until all pages are
    * mapped.
    * Since we are adding new mappings, there is no need to flush TLBs.
    * Return 0 on success.
    */
```

```
u64 cnt = DIV_ROUND_UP(len, PAGE_SIZE);
u64 idx = 0;
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
ptp_t *phys_page;
pte_t *pte;
vaddr_t cur_va;
paddr_t cur_pa;
int ret;

while (idx < cnt) {
        cur_va = va + idx * PAGE_SIZE;
        cur_pa = pa + idx * PAGE_SIZE;

        l0_ptp = (ptp_t *)pgtbl;
        ret = get_next_ptp(l0_ptp, L0, cur_va, &l1_ptp, &pte, true, rss);
        if (ret < 0) {
                return ret;
        }

        ret = get_next_ptp(l1_ptp, L1, cur_va, &l2_ptp, &pte, true, rss);
        if (ret < 0) {
                return ret;
        } else if (ret == BLOCK_PTP) {
                idx += L1_PER_ENTRY_PAGES;
                continue;
        }

        ret = get_next_ptp(l2_ptp, L2, cur_va, &l3_ptp, &pte, true, rss);
        if (ret < 0) {
                return ret;
        } else if (ret == BLOCK_PTP) {
                idx += L2_PER_ENTRY_PAGES;
                continue;
        }

        idx += L3_PER_ENTRY_PAGES;
        pte = &(l3_ptp->ent[GET_L3_INDEX(cur_va)]);
        set_pte_flags(pte, flags, kind);
        pte->l3_page.is_page = 1;
        pte->l3_page.is_valid = 1;
        pte->l3_page.pfn = cur_pa >> L3_INDEX_SHIFT;
}
/* LAB 2 TODO 4 END */
```

```
/* LAB 2 TODO 4 BEGIN */
/*
    * Hint: Walk through each level of page table using `get_next_ptp`,
    * mark the final level pte as invalid. Iterate until all pages are
    * unmapped.
    * You don't need to flush tlb here since tlb is now flushed after
    * this function is called.
```

```
    * Return 0 on success.
    */
u64 cnt = DIV_ROUND_UP(len, PAGE_SIZE);
u64 idx = 0;
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
ptp_t *phys_page;
pte_t *pte;
vaddr_t cur_va;
int ret;

while (idx < cnt) {
        cur_va = va + idx * PAGE_SIZE;

        l0_ptp = (ptp_t *)pgtbl;
        ret = get_next_ptp(l0_ptp, L0, cur_va, &l1_ptp, &pte, false, rss);
        if (ret < 0) {
                return ret;
        }

        ret = get_next_ptp(l1_ptp, L1, cur_va, &l2_ptp, &pte, false, rss);
        if (ret < 0) {
                return ret;
        } else if (ret == BLOCK_PTP) {
                idx += L1_PER_ENTRY_PAGES;
                pte->pte = PTE_DESCRIPTOR_INVALID;
                continue;
        }

        ret = get_next_ptp(l2_ptp, L2, cur_va, &l3_ptp, &pte, false, rss);
        if (ret < 0) {
                return ret;
        } else if (ret == BLOCK_PTP) {
                idx += L2_PER_ENTRY_PAGES;
                pte->pte = PTE_DESCRIPTOR_INVALID;
                continue;
        }

        ret = get_next_ptp(l3_ptp, L3, cur_va, &phys_page, &pte, false, rss);
        if (ret < 0) {
                return ret;
        } else {
                idx += L3_PER_ENTRY_PAGES;
                pte->pte = PTE_DESCRIPTOR_INVALID;
                recycle_pgtable_entry(l0_ptp, l1_ptp, l2_ptp, l3_ptp, cur_va, rss);
        }
}
/* LAB 2 TODO 4 END */
```

```
/* LAB 2 TODO 4 BEGIN */
/*
    * Hint: Walk through each level of page table using `get_next_ptp`,
```

```
    * modify the permission in the final level pte using `set_pte_flags`.
    * The `kind` argument of `set_pte_flags` should always be `USER_PTE`.
    * Return 0 on success.
    */
u64 cnt = DIV_ROUND_UP(len, PAGE_SIZE);
u64 idx = 0;
ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
ptp_t *phys_page;
pte_t *pte;
vaddr_t cur_va;
int ret;

while (idx < cnt) {
        cur_va = va + idx * PAGE_SIZE;

        l0_ptp = (ptp_t *)pgtbl;
        ret = get_next_ptp(l0_ptp, L0, cur_va, &l1_ptp, &pte, false, NULL);
        if (ret < 0) {
                return ret;
        }

        ret = get_next_ptp(l1_ptp, L1, cur_va, &l2_ptp, &pte, false, NULL);
        if (ret < 0) {
                return ret;
        } else if (ret == BLOCK_PTP) {
                idx += L1_PER_ENTRY_PAGES;
                continue;
        }

        ret = get_next_ptp(l2_ptp, L2, cur_va, &l3_ptp, &pte, false, NULL);
        if (ret < 0) {
                return ret;
        } else if (ret == BLOCK_PTP) {
                idx += L2_PER_ENTRY_PAGES;
                continue;
        }

        ret = get_next_ptp(l3_ptp, L3, cur_va, &phys_page, &pte, false, NULL);
        if (ret < 0) {
                return ret;
        } else {
                idx += L3_PER_ENTRY_PAGES;
                set_pte_flags(pte, flags, USER_PTE);
        }
}
/* LAB 2 TODO 4 END */
```

> 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）

为了支持写时拷贝，需要在页表项中将 `AP`（Access Permissions）字段配置为只读。

只读页内发生写操作时，会触发缺页异常，操作系统检测到异常后，会重新分配一个物理页，将共享页的内容拷贝至新的物理页中，并将访问权限配置为可读可写，然后更新页表映射。

> 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。

一方面，为内核分配的大页通常不会被完全使用，从而产生大量内部碎片，造成内存浪费。

另一方面，粗粒度的页表缺少详细的权限控制，降低了内存访问的安全性。

> 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。

首先调用 `get_pages` 函数申请一个物理页作为根页表，然后调用 `map_range_in_pgtbl_kernel` 函数分段映射内核空间，最后将根页表的物理地址通过 `msr` 指令写入 `ttbr1_el1` 寄存器。值得注意的是，为了保证 C 函数的正常调用，内核栈也需要重新映射。

```
/* Remap kernel space into 4KB pages. */
#define PHYSICAL_START  (0x0UL)
#define DEVICE_START    (0x3F000000UL)
#define DEVICE_END      (0x40000000UL)
#define PHYSICAL_END    (0x80000000UL)
#define KERNEL_BIAS     (0xffffff0000000000)
    volatile u64 ttbr1_el1 = get_pages(0);
    map_range_in_pgtbl_kernel((void *)ttbr1_el1, KSTACKx_ADDR(0),
            virt_to_phys(cpu_stacks[0]), CPU_STACK_SIZE, VMR_READ | VMR_WRITE);
    map_range_in_pgtbl_kernel((void *)ttbr1_el1, KERNEL_BIAS + PHYSICAL_START,
            PHYSICAL_START, DEVICE_START - PHYSICAL_START, VMR_EXEC);
    map_range_in_pgtbl_kernel((void *)ttbr1_el1, KERNEL_BIAS + DEVICE_START,
            DEVICE_START, DEVICE_END - DEVICE_START, VMR_DEVICE);
    map_range_in_pgtbl_kernel((void *)ttbr1_el1, KERNEL_BIAS + DEVICE_END,
            DEVICE_END, PHYSICAL_END - DEVICE_END, VMR_DEVICE);
    volatile u64 phys_addr = virt_to_phys((void *)ttbr1_el1);
    asm volatile("msr ttbr1_el1, %0" : : "r" (phys_addr));
    flush_tlb_all();
    kinfo("[ChCore] kernel remap finished\n");
#undef PHYSICAL_START
#undef DEVICE_START
#undef DEVICE_END
#undef PHYSICAL_END
#undef KERNEL_BIAS
```

> 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。

调用 `handle_trans_fault` 函数处理地址翻译错误，并将返回值存入 `ret` 即可。

```
/* LAB 2 TODO 5 BEGIN */
ret = handle_trans_fault(current_thread->vmspace, fault_addr);
/* LAB 2 TODO 5 END */
```

> 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。
>
> 提示:
>
> - 一个虚拟地址空间所包含的 VMR 通过 rb_tree 的数据结构保存在 `vmspace` 结构体的 `vmr_tree` 字段
> - 可以使用 `kernel/include/common/rbtree.h` 中定义的 `rb_search`、`rb_entry` 等函数或宏来对 rb_tree 进行搜索或操作

利用 `rb_search` 函数高效查找虚拟地址所在的 `node`，然后利用 `rb_entry` 宏取出相应的 `vmr`。

```
/* LAB 2 TODO 6 BEGIN */
/* Hint: Find the corresponding vmr for @addr in @vmspace */
struct rb_node *node;
struct vmregion *vmr = NULL;

node = rb_search(&vmspace->vmr_tree, (const void *)addr, cmp_vmr_and_va);
if (node != NULL) {
        vmr = rb_entry(node, struct vmregion, tree_node);
}

return vmr;
/* LAB 2 TODO 6 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。

若物理页尚未分配，则调用 `get_pages` 函数申请一个物理页，然后将其内容清零。

```
/* LAB 2 TODO 7 BEGIN */
/* Hint: Allocate a physical page and clear it to 0. */
vaddr_t va = get_pages(0);
pa = virt_to_phys(va);
BUG_ON(pa == 0);
memset(va, 0, PAGE_SIZE);
/* LAB 2 TODO 7 END */
```

无论物理页是否已分配，都要调用 `map_range_in_pgtbl` 函数配置页表映射。

```
/* LAB 2 TODO 7 BEGIN */
ret = map_range_in_pgtbl(vmspace->pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss);
/* LAB 2 TODO 7 END */
```


```
/* LAB 2 TODO 7 BEGIN */
ret = map_range_in_pgtbl(vmspace->pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss);
/* LAB 2 TODO 7 END */
```

> 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。

首先，我们定义 `lru_queue` 数据结构，用于记录最近使用的物理页，另外使用 `lru_count` 变量维护队列中已有物理页的数量。

```
#define LRU_CAPACITY 64
static vaddr_t lru_queue[LRU_CAPACITY];
static size_t lru_count = 0;
```

然后，我们需要实现 `swap_in` 和 `swap_out` 两个函数。物理页换出时，我们将其内容写入磁盘，并将其地址记录在页表项中，物理页换入时，我们从页表项中查询磁盘地址，然后将相应内容读入物理页，并重新配置页表映射。值得注意的是，在修改页表项之后，我们需要调用 `flush_tlb_by_range` 函数刷新 TLB。

```
static void swap_in(struct vmspace *vmspace, vaddr_t va)
{
        struct vmregion *vmr;
        vmr_prop_t perm;
        vaddr_t disk_va;
        pte_t *pte;
        paddr_t pa;

        query_in_pgtbl(vmspace->pgtbl, va, &pa, &pte);
        pa = virt_to_phys(get_pages(0));
        vmr = find_vmr_for_va(vmspace, va);
        perm = vmr->perm;
        map_range_in_pgtbl(vmspace->pgtbl, va, pa, PAGE_SIZE, perm, NULL);
        disk_va = pte->pte;
        memcpy(va, disk_va, PAGE_SIZE);
        free_pages(disk_va);
        flush_tlb_by_range(vmspace, va, PAGE_SIZE);
}
```

```
static void swap_out(struct vmspace *vmspace, vaddr_t va)
{
        vaddr_t disk_va;
        pte_t *pte;
        paddr_t pa;

        query_in_pgtbl(vmspace->pgtbl, va, &pa, &pte);
        disk_va = get_pages(0);
        pte->pte = disk_va;
        memcpy(disk_va, va, PAGE_SIZE);
        free_pages(va);
        flush_tlb_by_range(vmspace, va, PAGE_SIZE);
}
```

最后，我们实现 `lru_update` 函数，用于更新物理页的访问信息。当物理页已在队列中时，我们将其移至队尾，否则将其插入队尾，若此时队列已满，则将位于队首的物理页换出。

```
static void lru_update(struct vmspace *vmspace, vaddr_t va)
{
        int idx;

        for (idx = 0; idx < lru_count; idx++) {
                if (lru_queue[idx] == va) {
                        break;
                }
        }

        if (idx == lru_count) {
                if (lru_count == LRU_CAPACITY) {
                        swap_out(vmspace, lru_queue[0]);
                        for (int i = 0; i < LRU_CAPACITY - 1; i++) {
                                lru_queue[i] = lru_queue[i + 1];
                        }
                        --lru_count;
                }
                lru_queue[lru_count] = va;
                ++lru_count;
        } else {
                for (int pos = idx; pos < lru_count - 1; pos++) {
                        lru_queue[pos] = lru_queue[pos + 1];
                }
        }
}
```

至此，运行 `make qemu` 可以正常进入 `shell`，运行 `make grade` 可以通过所有测试。

```
 ______     __  __     ______     __  __     ______     __         __
/\  ___\   /\ \_\ \   /\  ___\   /\ \_\ \   /\  ___\   /\ \       /\ \
\ \ \____  \ \  __ \  \ \___  \  \ \  __ \  \ \  __\   \ \ \____  \ \ \____
 \ \_____\  \ \_\ \_\  \/\_____\  \ \_\ \_\  \ \_____\  \ \_____\  \ \_____\
  \/_____/   \/_/\/_/   \/_____/   \/_/\/_/   \/_____/   \/_____/   \/_____/

Welcome to ChCore shell!
```

```
===============
Grading lab 2...(may take 10 seconds)
GRADE: Allocate & free order 0: 5
GRADE: Allocate & free each order: 5
GRADE: Allocate & free all orders: 5
GRADE: Allocate & free all memory: OK: 5
GRADE: kmalloc: 10
GRADE: Map & unmap one page: 10
GRADE: Map & unmap multiple pages: 10
GRADE: Map & unmap huge range: 20
GRADE: Page fault: 30
===============
Score: 100/100
```