Lab2-Report

思考题 1：请思考多级页表相比单级页表带来的优势和劣势（如果有的话），并计算在 AArch64页表中分别以 4KB 粒度和 2MB 粒度映射 0～4GB 地址范围所需的物理内存大小（或页表页数量）。

优势：多级页表能有效压缩页表的大小，可以节省内存资源

劣势：需要多次访存，复杂度可能比较高

若选择4KB 粒度时：

$$
\begin{gathered}
\text{每张页表页管理的页表条目数为} 4KB/8byte = 512\text{个} \\
\text{一个} L3\text{页表所映射的物理内存大小为} 512 * 4KB = 2MB \\
\text{一个} L2\text{页表所映射的物理内存大小为} 512 * 2MB = 1GB \\
\text{所以总共需要} 1\text{个} L0\text{页表，} 1\text{个} L1\text{页表，} 4GB/1GB = 4\text{个} L2\text{页表，以及} 4 * 512 = 2048\text{个} L3\text{页表} \\
\text{总计} 1 + 1 + 4 + 2048 = 2054\text{个页表}
\end{gathered}
$$

若选择2MB 粒度时：

$$
\begin{gathered}
\text{每张页表页管理的页表条目数为} 4KB/8byte = 512\text{个} \\
\text{以} 2MB\text{粒度映射，也就是说使用} L2block\text{的大页进行映射} \\
\text{则一张} L2\text{页表所映射的物理内存大小为} 512 * 2MB = 1G \\
\text{因而总共需要} 4\text{个} L2\text{页表，} 1\text{个} L1\text{页表，} 1\text{个} L0\text{页表} \\
\text{总共} 1 + 1 + 4 = 6\text{个页表}
\end{gathered}
$$

练习题 2：请在 init_boot_pt 函数的 LAB 2 TODO 1 处配置内核高地址页表 （ boot_ttbr1_l0 、boot_ttbr1_l1 和 boot_ttbr1_l2 ），以 2MB 粒度映射。

这里我们要做的，是将虚拟内存高地址映射到内核所在的物理内存。因此在页表中的虚拟地址均需要加上`KERNEL_VADDR`以将其转换为内核态的高地址。

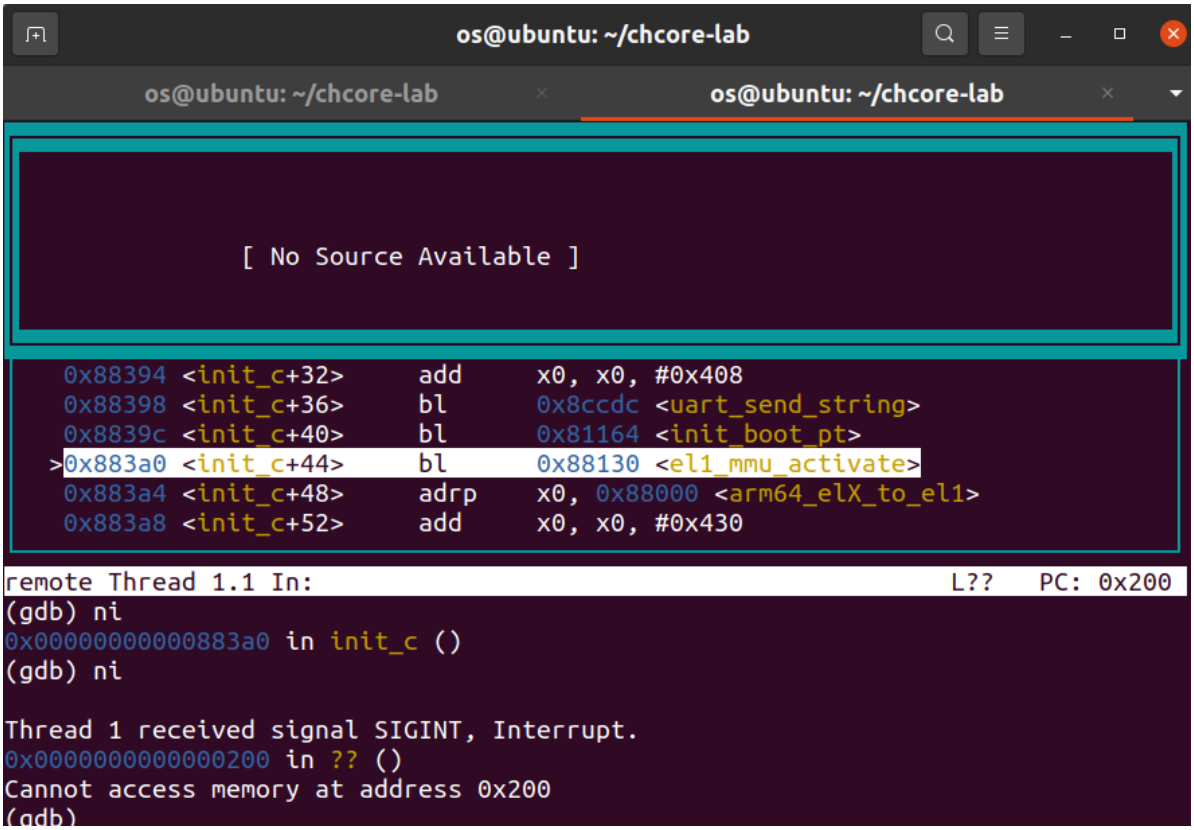
```
/* TTBR1_EL1 0-1G */
/* LAB 2 TODO 1 BEGIN */
/* Step 1: set L0 and L1 page table entry */
vaddr = PHYSMEM_START;
boot_ttbr1_l0[GET_L0_INDEX(KERNEL_VADDR + vaddr)] = ((u64)boot_ttbr1_l1) | IS_TABLE
                                                  | IS_VALID | NG;
boot_ttbr1_l1[GET_L1_INDEX(KERNEL_VADDR + vaddr)] = ((u64)boot_ttbr1_l2) | IS_TABLE
                                                  | IS_VALID | NG;

/* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
for (; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(KERNEL_VADDR + vaddr)] =
                (vaddr) /* low mem, va = pa */
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | INNER_SHARABLE /* Sharebility */
                | NORMAL_MEMORY /* Normal memory */
                | IS_VALID;
}
/* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(KERNEL_VADDR + vaddr)] =
                (vaddr) /* low mem, va = pa */
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | DEVICE_MEMORY /* Device memory */
                | IS_VALID;
}
/* LAB 2 TODO 1 END */
```

思考题 3：请思考在 init_boot_pt 函数中为什么还要为低地址配置页表，并尝试验证自己的解释。

`el1_mmu_activate` 函数启用了MMU，使可以通过虚拟地址访问内存。但在通过 `start_kernel` 函数跳转到高地址之前，代码仍"运行在"虚拟内存的低地址中，所以也需要为低地址配置页表。

验证：我们可以将 `init_boot_pt` 函数中为低地址配置页表的部分代码注释掉，运用GDB进行调试，可以发现在运行到 `el1_mmu_activate` 函数之后无法继续正常运行：由于没有为低地址配置页表，在启用 MMU 后内核"运行在"虚拟内存的低地址中，会产生地址翻译错误，进而尝试跳转到异常处理函数，而此时我们也没有设置异常向量表，因此执行流会 在 `0x200` 地址无限循环。



练习题 4：完成 kernel/mm/buddy.c 中的 split_page 、 buddy_get_pages 、 merge_page 和 buddy_free_pages 函数中的 LAB 2 TODO 2 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_page(struct phys_mem_pool *pool, u64 order,
                               struct page *page)
{
        /* LAB 2 TODO 2 BEGIN */
        /*
         * Hint: Recursively put the buddy of current chunk into
         * a suitable free list.
         */
        if(page->order == order){
                return page;
        }
        list_del(&(page->node));
        pool->free_lists[page->order].nr_free--;
        page->order--;
        pool->free_lists[page->order].nr_free += 2;
        struct page *buddy_page = get_buddy_chunk(pool, page);
        buddy_page->order = page->order;
        buddy_page->allocated = 0;
        buddy_page->pool = pool;
        list_add(&buddy_page->node, &(pool->free_lists[page->order].free_list));
        list_add(&page->node, &(pool->free_lists[page->order].free_list));

        return split_page(pool, order, page);

        /* LAB 2 TODO 2 END */
}

struct page *buddy_get_pages(struct phys_mem_pool *pool, u64 order)
```

```
{
        /* LAB 2 TODO 2 BEGIN */
        /*
         * Hint: Find a chunk that satisfies the order requirement
         * in the free lists, then split it if necessary.
         */
        if(order > BUDDY_MAX_ORDER - 1){
                return NULL;
        }
        u64 current_order = order;
        while(list_empty(&(pool->free_lists[current_order].free_list))){
                current_order++;
                if(current_order > BUDDY_MAX_ORDER - 1){
                        return NULL;
                }
        }
        struct page *page = list_entry(pool-
>free_lists[current_order].free_list.next, struct page, node);
        if(current_order > order){
                page = split_page(pool, order, page);
        }

        pool->free_lists[order].nr_free--;
        list_del(&(page->node));
        page->allocated = 1;

        return page;

        /* LAB 2 TODO 2 END */
}

static struct page *merge_page(struct phys_mem_pool *pool, struct page *page)
{
        /* LAB 2 TODO 2 BEGIN */
        /*
         * Hint: Recursively merge current chunk with its buddy
         * if possible.
         */
        if(page->order >= BUDDY_MAX_ORDER - 1){
                return page;
        }
        struct page *buddy_page = get_buddy_chunk(pool, page);
        if(buddy_page == NULL || buddy_page->allocated){
                return page;
        }
        struct page *merged_page  = NULL;
        /* Compare the address of the chunk to determine address of merged_page
*/
        if((u64)page_to_virt(page) < (u64)page_to_virt(buddy_page)){
                merged_page = page;
        }
        else{
                merged_page = buddy_page;
        }
        pool->free_lists[page->order].nr_free -= 2;
        list_del(&(page->node));
        list_del(&(buddy_page->node));
        merged_page->order++;
```

```
        pool->free_lists[merged_page->order].nr_free++;
        list_add(&(merged_page->node), &(pool->free_lists[merged_page-
>order].free_list));

        return merge_page(pool, merged_page);

        /* LAB 2 TODO 2 END */
}

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

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

练习题 5：完成 kernel/arch/aarch64/mm/page_table.c 中的 query_in_pgtbl 、map_range_in_pgtbl 、 unmap_range_in_pgtbl 函数中的 LAB 2 TODO 3 部分， 分别实现页表查询、映射、取消映射操作。

```
/*
 * Translate a va to pa, and get its pte for the flags
 */
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
{
        /* LAB 2 TODO 3 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 *cur_ptp = (ptp_t *)pgtbl;
        ptp_t *next_ptp;
        pte_t *pte;

        for(int i = 0; i <= 3; i++){
                int type = get_next_ptp(cur_ptp, i, va, &next_ptp, &pte, false);
                if(type == BLOCK_PTP){
                        *entry = pte;
                        switch (i)
                        {
                        case 1:
                                *pa = virt_to_phys((vaddr_t) next_ptp) +
GET_VA_OFFSET_L1(va);
                                break;
```

```
                        case 2:
                                *pa = virt_to_phys((vaddr_t) next_ptp) +
GET_VA_OFFSET_L2(va);
                                break;

                        default:
                                break;
                        }

                        return 0;
                }
                else{
                        if(type == NORMAL_PTP){
                                cur_ptp = next_ptp;
                        }
                        else{
                                return type;
                        }
                }
        }

        *entry = pte;
        *pa = virt_to_phys((vaddr_t) next_ptp) + GET_VA_OFFSET_L3(va);

        return 0;

        /* LAB 2 TODO 3 END */
}

int map_range_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
                       vmr_prop_t flags)
{
        /* LAB 2 TODO 3 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.
         */
        ptp_t *ptp_0, *ptp_1, *ptp_2, *ptp_3;
        ptp_0 = (ptp_t *)pgtbl;
        pte_t *pte_0, *pte_1, *pte_2, *pte_3;
        int page_num = (len - 1) / PAGE_SIZE + 1;

        for(int i = 0; i < page_num; i++){
                len -= PAGE_SIZE;
                int type = get_next_ptp(ptp_0, 0, va, &ptp_1, &pte_0, true);
                if(type < 0){
                        return type;
                }

                type = get_next_ptp(ptp_1, 1, va, &ptp_2, &pte_1, true);
                if(type < 0){
                        return type;
                }

                type = get_next_ptp(ptp_2, 2, va, &ptp_3, &pte_2, true);
                if(type < 0){
```

```
                        return type;
                }

                int index = GET_L3_INDEX(va);
                pte_3 = &(ptp_3->ent[index]);
                pte_3->l3_page.is_page = 1;
                pte_3->l3_page.is_valid = 1;
                pte_3->l3_page.pfn = pa >> PAGE_SHIFT;
                set_pte_flags(pte_3, flags, USER_PTE);

                va += PAGE_SIZE;
                pa += PAGE_SIZE;
        }

        return 0;

        /* LAB 2 TODO 3 END */
}

int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len)
{
        /* LAB 2 TODO 3 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.
         */
        ptp_t *ptp_0, *ptp_1, *ptp_2, *ptp_3;
        ptp_0 = (ptp_t *)pgtbl;
        pte_t *pte_0, *pte_1, *pte_2, *pte_3;
        int page_num = (len - 1) / PAGE_SIZE + 1;

        for(int i = 0; i < page_num; i++){
                int type = get_next_ptp(ptp_0, 0, va, &ptp_1, &pte_0, false);
                if(type < 0){
                        return type;
                }

                type = get_next_ptp(ptp_1, 1, va, &ptp_2, &pte_1, false);
                if(type < 0){
                        return type;
                }

                type = get_next_ptp(ptp_2, 2, va, &ptp_3, &pte_2, false);
                if(type < 0){
                        return type;
                }

                ptp_t *next_ptp;
                type = get_next_ptp(ptp_3, 3, va, &next_ptp, &pte_3, false);
                if(type < 0){
                        return type;
                }
                pte_3->l3_page.is_valid = 0;

                va += PAGE_SIZE;
        }
```

```
        return 0;

        /* LAB 2 TODO 3 END */
}
```

练习题 6：完成 kernel/arch/aarch64/mm/page_table.c 中的 map_range_in_pgtbl_huge 和 unmap_range_in_pgtbl_huge 函数中的 LAB 2 TODO 4 部分，实现大页（2MB、1GB 页） 支持。

```
int map_range_in_pgtbl_huge(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
                            vmr_prop_t flags)
{
        /* LAB 2 TODO 4 BEGIN */
        ptp_t *ptp_0, *ptp_1, *ptp_2, *ptp_3;
        ptp_0 = (ptp_t *)pgtbl;
        pte_t *pte_0, *pte_1, *pte_2, *pte_3;

        while(true){
                if(len >= (1 << 30)){
                        len -= (1 << 30);
                        int type = get_next_ptp(ptp_0, 0, va, &ptp_1, &pte_0,
true);
                        if(type < 0){
                                return type;
                        }

                        int index = GET_L1_INDEX(va);
                        pte_1 = &(ptp_1->ent[index]);
                        pte_1->l1_block.is_valid = 1;
                        pte_1->l1_block.is_table = 0;
                        pte_1->l1_block.pfn = pa >> 30;
                        set_pte_flags(pte_1, flags, USER_PTE);

                        va += (1 << 30);
                        pa += (1 << 30);
                }
                else{
                        if(len >= (2 << 20)){
                                len -= (2 << 20);
                                int type = get_next_ptp(ptp_0, 0, va, &ptp_1,
&pte_0, true);
                                if(type < 0){
                                        return type;
                                }

                                type = get_next_ptp(ptp_1, 1, va, &ptp_2, &pte_1,
true);
                                if(type < 0){
                                        return type;
                                }

                                int index = GET_L2_INDEX(va);
                                pte_2 = &(ptp_2->ent[index]);
                                pte_2->l2_block.is_table = 0;
                                pte_2->l2_block.is_valid = 1;
                                pte_2->l2_block.pfn = pa >> 21;
```

```
                                set_pte_flags(pte_2, flags, USER_PTE);

                                va += (2 << 20);
                                pa += (2 << 20);
                        }
                        else{
                                if(len > 0){
                                        len -= PAGE_SIZE;
                                        int type = get_next_ptp(ptp_0, 0, va,
&ptp_1, &pte_0, true);
                                        if(type < 0){
                                                return type;
                                        }

                                        type = get_next_ptp(ptp_1, 1, va, &ptp_2,
&pte_1, true);
                                        if(type < 0){
                                                return type;
                                        }

                                        type = get_next_ptp(ptp_2, 2, va, &ptp_3,
&pte_2, true);
                                        if(type < 0){
                                                return type;
                                        }

                                        int index = GET_L3_INDEX(va);
                                        pte_3 = &(ptp_3->ent[index]);
                                        pte_3->l3_page.is_page = 1;
                                        pte_3->l3_page.is_valid = 1;
                                        pte_3->l3_page.pfn = pa >> PAGE_SHIFT;
                                        set_pte_flags(pte_3, flags, USER_PTE);

                                        va += PAGE_SIZE;
                                        pa += PAGE_SIZE;
                                }
                                else{
                                        return 0;
                                }
                        }
                }
        }

        /* LAB 2 TODO 4 END */
}

int unmap_range_in_pgtbl_huge(void *pgtbl, vaddr_t va, size_t len)
{
        /* LAB 2 TODO 4 BEGIN */
        ptp_t *ptp_0, *ptp_1, *ptp_2, *ptp_3;
        ptp_0 = (ptp_t *)pgtbl;
        pte_t *pte_0, *pte_1, *pte_2, *pte_3;

        while(true){
                if(len >= (1 << 30)){
                        len -= (1 << 30);
                        int type = get_next_ptp(ptp_0, 0, va, &ptp_1, &pte_0,
true);
```

```
                        if(type < 0){
                                return type;
                        }
                        type = get_next_ptp(ptp_1, 1, va, &ptp_2, &pte_1, true);
                        if(type < 0){
                                return type;
                        }
                        pte_1->l1_block.is_valid = 0;
                        va += (1 << 30);
                }
                else{
                        if(len >= (2 << 20)){
                                len -= (2 << 20);
                                int type = get_next_ptp(ptp_0, 0, va, &ptp_1,
&pte_0, true);
                                if(type < 0){
                                        return type;
                                }

                                type = get_next_ptp(ptp_1, 1, va, &ptp_2, &pte_1,
true);
                                if(type < 0){
                                        return type;
                                }

                                type = get_next_ptp(ptp_2, 2, va, &ptp_3, &pte_2,
true);
                                if(type < 0){
                                        return type;
                                }
                                pte_2->l2_block.is_valid = 0;
                                va += (2 << 20);
                        }
                        else{
                                if(len > 0){
                                        len -= PAGE_SIZE;
                                        int type = get_next_ptp(ptp_0, 0, va,
&ptp_1, &pte_0, false);
                                        if(type < 0){
                                                return type;
                                        }

                                        type = get_next_ptp(ptp_1, 1, va, &ptp_2,
&pte_1, false);
                                        if(type < 0){
                                                return type;
                                        }

                                        type = get_next_ptp(ptp_2, 2, va, &ptp_3,
&pte_2, false);
                                        if(type < 0){
                                                return type;
                                        }

                                        ptp_t *next_ptp;
                                        type = get_next_ptp(ptp_3, 3, va,
&next_ptp, &pte_3, false);
                                        if(type < 0){
```

```
                                        return type;
                                }
                                pte_3->l3_page.is_valid = 0;

                                va += PAGE_SIZE;
                        }
                        else{
                                return 0;
                        }
                }
        }
}

        /* LAB 2 TODO 4 END */
}
```

思考题 7：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，并在发生缺页异常（实际上是 permission fault）时如何处理。

支持写时拷贝时，需要配置AP字段来控制访问权限，将页表配置为只读权限。

在发生缺页异常后，CPU会将控制流传递给操作系统预先设置的缺页异常处理函数，操作系统会在物理内存中将缺页异常对应的物理页重新拷贝一份，并且将新拷贝的物理页以可读可写的方式重新映射给出发异常的应用程序，此后再恢复应用程序的执行。

思考题 8：为了简单起见，在 ChCore 实验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。

1. 粗粒度的页表可能导致因未使用整个大页而造成物理内存的资源浪费，产生更多的内部碎片，还会增加操作系统管理内存的复杂度。
2. 粗粒度页表可能会使得访问权限划分不够精细，造成不安全访问。例如.data段(可读可写)和.text段((可读可执行，但不可写)都放在同一个具有读写权限的页面内，甚至可能导致.text的代码也可以被修改，造成安全隐患。

519021910594

陶昱丞